DOCUMENT RESUME

ED 330 310 IR 014 925

AUTHOR Yoder, Sharon Burrowes
TITLE Introduction to Programming Using Terrapin Logo for the Macintosh.
INSTITUTION International Society for Technology in Education, Eugene, OR.
REPORT NO ISBN-0-924667-84-2
PUB DATE 91
NOTE 213p.
AVAILABLE FROM International Society for Technology in Education, 1787 Agate Street, Eugene, OR 97403-1923.
PUB TYPE Guides - Classroom Use - Instructional Materials (For Learner) (051) -- Guides - Classroom Use - Teaching Guides (For Teacher) (052)

EDRS PRICE MF01 Plus Postage. PC Not Available from EDRS.
DESCRIPTORS Authoring Aids (Programing); Computer Assisted Instruction; Computer Graphics; Computer Software; Elementary School Teachers; Elementary Secondary Education; Microcomputers; *Programing; Programing Languages; Secondary School Teachers
IDENTIFIERS *Apple Macintosh; *LOGO Programing Language

ABSTRACT

This manual has a dual purpose: to teach about programming in the Logo language for teachers, and to teach Logo in a structured manner to students. The publication is divided into 41 sections, each of which focuses on a particular Logo word or idea, and is designed to give both teachers and students the opportunity to experiment with ideas while extending their knowledge of Logo. Some of the subjects covered in the individual chapters include: (1) getting started with Logo for the Macintosh computer; (2) using the Repeat function; (3) shapes and STAMP; (4) defining new shapes, putting text on the screen, and mixing text and graphics; (5) writing procedures; (6) designing programs; (7) managing workspace and files; (8) using pen modes, windows, and multiple turtles; (9) Logo grammar; (10) animation, creating definite loops, stamping areas, interactive programming, and working with files; (11) procedure inputs; (12) Logo arithmetic and mathematics; (13) outputs; and (14) useful graphics and text reporters. Most of the chapters provide tips and techniques as well as project suggestions. It is suggested that the examples in each chapter be experimented with until the concept being introduced is understood. A quick reference card of all the Terrapin Logo primitives is provided at the end of the manual, and major subject areas and Logo commands are listed in the index. (DB)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

About the Author

Sharon Yoder has taught mathematics and computer science at the junior high and high school level for 15 years. Her most recent public school experience was as a secondary computer science teacher and a computer coordinator involved in developing system-wide computer curriculum and in planning teacher inservice training.

In addition, she has taught mathematics, computer science, and computer education at a number of universities in northeastern Ohio, including Kent State University, the University of Akron, and Cleveland State University. She has worked closely with the College of Education of Cleveland State University in developing their computer programming courses for teachers.

After a year as an Education Specialist for Logo Computer Systems, Inc., Sharon has returned to teaching. She is currently at the University of Oregon, where she teaches computer education courses.

For the past several years, she has conducted workshops and presented papers at conferences nation-wide, and has been involved in a number of book publishing projects, including the *Nudges* series. In addition, she has been a frequent contributor to *The Computing Teacher* and *Logo Exchange*. She is currently editor of *Logo Exchange* and column editor for The Logo Center in *The Computing Teacher*.

Project Editor: Ron Renchler
Cover Design: Percy Franklin
Production: Tamara Kidd

To order publications or to request a complete catalog contact:

INTERNATIONAL SOCIETY FOR TECHNOLOGY IN EDUCATION
1787 Agate Street, Eugene, Oregon 97403-1923
Phone 503/346-4414 FAX 503/346-5890
CompuServe: 70014,2117 Bitnet: ISTE@Oregon

ISTE is a non-profit organization with its main offices housed at the University of Oregon.

ISTE Publications

The text pages of this publication have been reproduced using recycled paper.

© 1990 ISTE
ISBN 0-924667-84-2

Introduction to Programming Using Terrapin Logo for the Macintosh

by Sharon Burrowes Yoder

© 1991 ISTE
ISBN 0-924667-84-2

Table of Contents

Introduction 1
Overview of This Book for Teachers 1
Overview of This Book for Learners 1

1. Getting Started with Logo for the Macintosh 3
   - Starting Logo 3
   - Drawing 4
   - Tips and Techniques 7
   - Project Suggestion 8

2. Using REPEAT 9
   - Wrapping 10
   - Adding Text to a Drawing 11
   - Tips and Techniques 12
   - Project Suggestions 12

3. Changing the Pen and RANDOM 13
   - Changing the Pen 13
   - Using RANDOM 14
   - Tips and Techniques 16
   - Project Suggestions 16

4. Shapes and STAMP 17
   - Snapping a Turtle Shape 17
   - Stamping the Turtle Shape 17
   - Drawing with Turtle Shapes 18
   - Tips and Techniques 19
   - Project Suggestions 19

5. Defining New Shapes 21
   - Tips and Techniques 23
   - Project Suggestions 24

6. Using FILLSH 25
   - Tips and Techniques 27
   - Project Suggestions 27

7. Putting Text on the Screen 29
   - Tips and Techniques 31
   - Project Suggestions 33

8. Mixing Text and Graphics 35
   - Tips and Techniques 37
   - Project Suggestions 37

9. Writing Procedures 39
   - Tips and Techniques 42
   - Project Suggestions 42

10. More Than One Procedure 43
Tips and Techniques 47
Project Suggestion 48

11. A Word About Designing Programs 49
Top Down Programming 49
Bottom Up Programming 49
Middle Out Programming 50
Polishing Your Program 50
Project Suggestions 51

12. Using PRINT 53
Tips and Techniques 56
Project Suggestions 57

13. Managing Your Workspace and Files 59
Commands to Manage Your Workspace 59
Tips for Managing Your Workspace 60
Working with Files 60
Workspace and Files 62
Project Suggestion 62

14. Using Pen Modes 63
PENREVERSE 63
PENMIX 66
Tips and Techniques 67
Project Suggestions 68

15. Using Windows 69
Tips and Techniques 73
Project Suggestions 73

16. Multiple Turtles 75
Tips and Techniques 78
Project Suggestion 79

17. Logo Grammar 81
Logo Objects 81
Logo Punctuation 83
Categories of Procedures 83
Summary 84
Tips and Techniques 84
Project Suggestion 85

18. Using READCHARACTER 87
Tips and Techniques 90
Project Suggestions 91

19. Indefinite Repetition 93
Tips and Techniques 95
Project Suggestions 88

20. Animation 97
Tips and Techniques 101
Project Suggestions 102

21. Creating Definite Loops 103
Tips and Techniques 105
Project Suggestions 106

22. Stamping Areas 107
Stamping Ovals 107
Stamping Rectangles 109
Stamping Arcs 111
Tips and Techniques 113
Project Suggestions 114

23. More Interactive Programming 115
Creating READWORD 116
Tips and Techniques 117
Project Suggestion 118

24. Working with Files 119
Tips and Techniques 120
Project Suggestions 120

25. More on Making Decisions 121
Tips and Techniques 122
Project Suggestion 123

26. Procedure Inputs 125
Looping Using a Procedure Input 127
Passing Values from the Keyboard to a Procedure Input 128
Creating a CASE Statement 129
Tips and Techniques 130
Project Suggestions 131

27. More on Multiple Turtles 133
Tips and Techniques 137
Project Suggestion 138

28. Using Inputs to Pass Values Among Procedures 139
Tips and Techniques 143
Project Suggestions 143

29. Incrementing 145
Using Incrementing 150
Embedded Recursion 152
Tips and Techniques 154
Project Suggestions 154

30. Logo Arithmetic and Math 155
Tips and Techniques 158
Project Suggestions 158

31. OUTPUT 159
Tips and Techniques 163
Project Suggestions 163

32. Some Useful Graphics Reporters 165
Project Suggestions 167

33. Some Useful Text Reporters 169
Project Suggestions 171

34. Time and the Mouse 173
Project Suggestion 174

35. Using MAKE 175
Tips and Techniques 178
Project Suggestions 179

36. Cartesian Primitives 181
Tips and Techniques 185
Project Suggestions 186

37. Predicates 187
Tips and Techniques 190
Project Suggestions 191

38. FIRST, BUTFIRST, LAST, and BUTLAST 193
Tips and Techniques 195
Project Suggestions 196

39. ITEM and COUNT 197
Tips and Techniques 200
Project Suggestions 200

40. Word, Sentence, and LIST 201
Project Suggestions 203

41. FPUT and LPUT 205
Tips and Techniques 206
Project Suggestion 206

Introduction

The Logo programming language was originally developed at MIT in 1968 by Seymour Papert and his colleagues. It was initially implemented on mainframe, timeshared computer systems. Those versions did not include the familiar turtle graphics that are now considered an integral part of Logo. In the early 1980s, Logo was implemented on microcomputers and began to appear in classrooms across the country. Since those early implementations, there have been many versions of Logo for many different computers. In this book, you will be learning about the version of Logo called *Terrapin Logo* for the Macintosh. Terrapin Software, Inc. was the one of the first companies to produce a version of Logo for microcomputer users. The version of Logo used in this book makes use of the Macintosh interface. It has beautiful graphics features based on the Macintosh capabilities and, of course, runs faster than Logo runs on many other microcomputers. However, most of what you will learn in this book about the Logo programming language applies to any version of Logo.

Overview of This Book for Teachers

The goal of this book is to teach about programming in the Logo language. However, you need to be aware that there is much more to Logo than just using it to learn programming. Logo encompasses a philosophy of learning, best explained in Seymour Papert's book *Mindstorms: Children, Computers, and Powerful Ideas*. You should balance your learning and/or teaching of Logo programming techniques with learning about and/or applying the Logo philosophy using sources such as the writings of Papert.

An *Introduction to Programming Using Terrapin Logo for the Macintosh* is a textbook-like source of information about Logo syntax and grammar. Although it is not as technical as a reference manual, it does contain detailed explanations of most Logo primitives. However, it goes further in that it discusses some computer science concepts, such as programming style, recursion, and control structures. As a teacher of Logo or trainer of Logo teachers, you need to be comfortable enough with Logo to help your students debug their programs. This book is designed to help you learn Logo syntax in a rather structured manner.

If you are a teacher of younger students, you should keep in mind, however, that this book is in *no way designed to model the way you should teach Logo in the classroom.* Your teaching should encompass the exploratory, discovery learning emphasized in the Logo philosophy. Most likely you will not want to teach a formal Logo lesson each day. On the other hand, if you are teaching older students, say in a computer science class, you may want to use a more structured approach, or perhaps combine the two approaches. Whatever way you teach Logo, you should be aware that there is nothing magical about the sequence in which topics are presented in this book. There are many appropriate paths available for learning about Logo.

Overview of This Book for Learners

This book is designed to teach you Logo in a rather structured manner. The major portion of the book is divided into small sections, each of which focuses on a particular Logo word or idea. You should not simply read the chapters in the book but rather should try out examples given in the text and experiment with them until you understand the concept being introduced. Each chapter ends with some ideas for "projects." These ideas are designed to give you an opportunity to experiment with the ideas you have learned and to extend your knowledge of Logo. You should take some time to work through some of these ideas before moving to the next chapter of the book.

At the end of this book you will find a "quick reference card" that includes a list of all the *Terrapin Logo* primitives. (The term "primitive" refers to a word Logo "knows.") You may want to mark your book so that you can easily locate this summary, or perhaps you should make copies of it for easy reference. It is not appropriate to memorize all the primitives you may want to use when working with *Terrapin Logo*. Rather, you should have on hand a reference in which to look up the information you need.

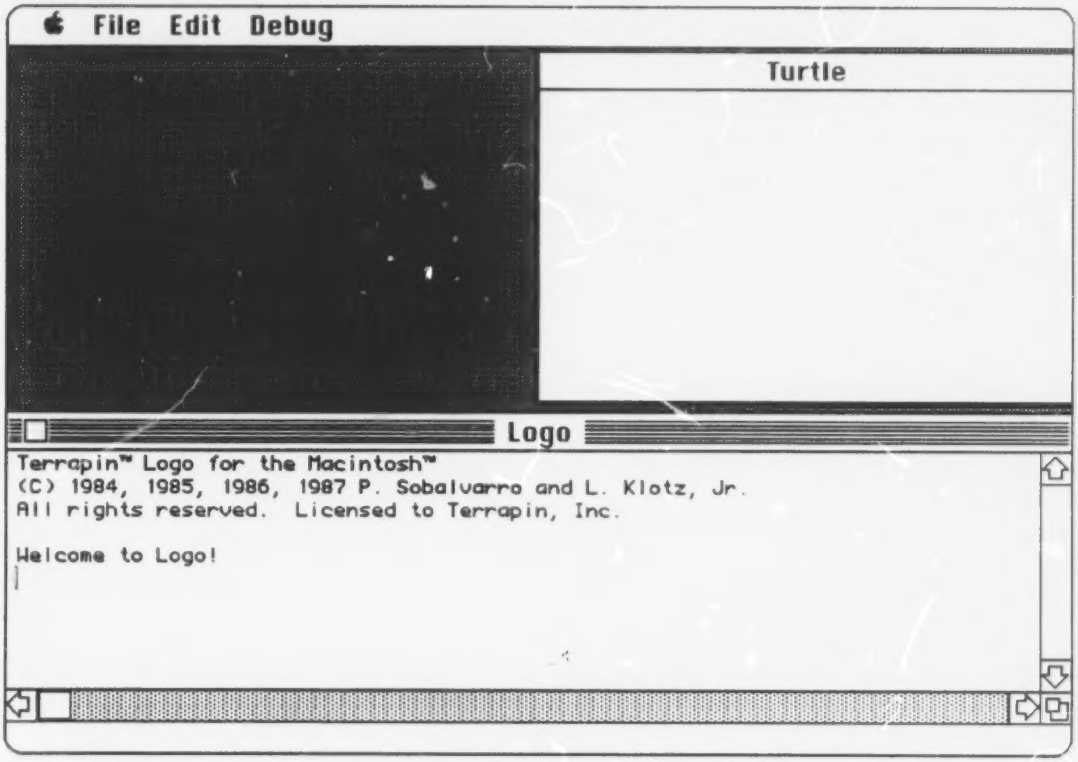
1. Getting Started with Logo for the Macintosh

Starting Logo

To begin you will need a disk containing *Terrapin Logo* for the Macintosh (the disk contains the Logo language). You can run *Terrapin Logo* using the System Folder that comes on the disk, or you can use your own System Folder. If you have a hard drive, you may want to move Logo onto your hard drive. Be sure you copy everything onto your hard drive *except* the System Folder. (Note: be sure you *do* copy the folder labelled "Logo System.") In any case, you should also have a formatted disk to use for saving or backing up your work.

Start *Terrapin Logo* as you would any Macintosh application. This book assumes you understand the Macintosh interface. If you have not used a Macintosh before, then you should read a book that introduces you to the Macintosh interface before continuing with this book.

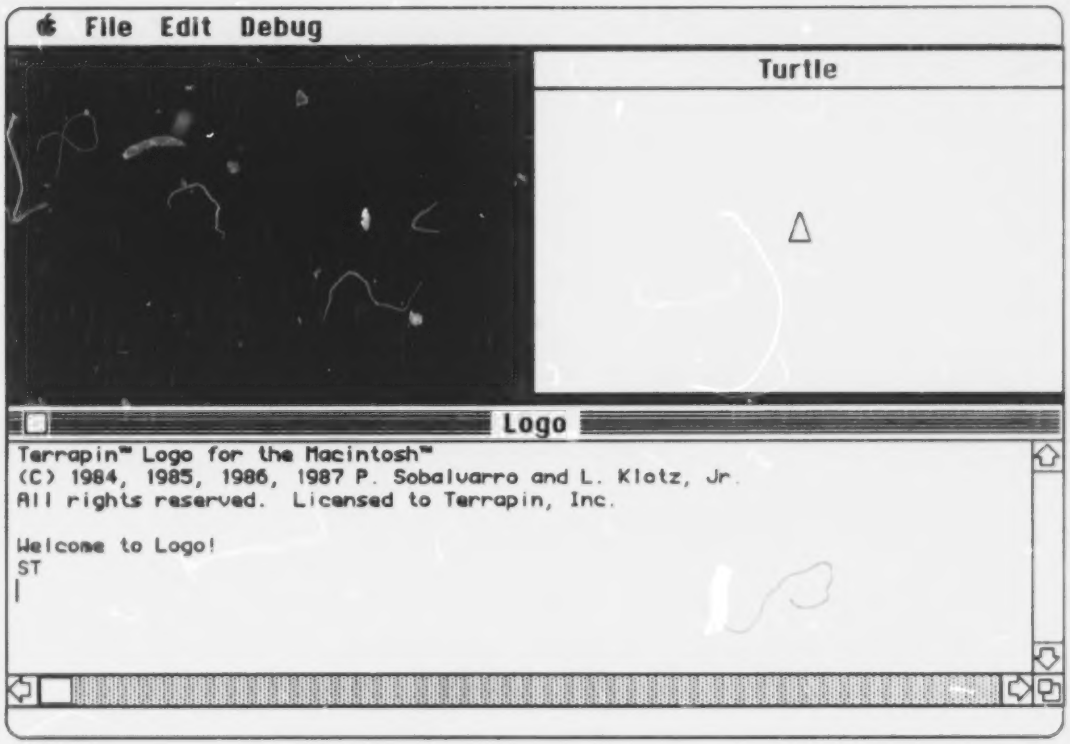
When you have started Logo, you see


File Edit Debug
Turtle
Logo
Terrapin™ Logo for the Macintosh™
(C) 1984, 1985, 1986, 1987 P. Sobalvarro and L. Klotz, Jr.
All rights reserved. Licensed to Terrapin, Inc.
Welcome to Logo!

Type

```
ST
```

(for ShowTurtle) and press the Enter key. Note: the Enter key is at the far right hand side of most Macintosh keyboards. **You must press Enter, *not* Return.** You then see:


File Edit Debug
Turtle
Logo
Terrapin™ Logo for the Macintosh™
(C) 1984, 1985, 1986, 1987 P. Sobalvarro and L. Klotz, Jr.
All rights reserved. Licensed to Terrapin, Inc.
Welcome to Logo!
ST

In the center of the Turtle Window is a small triangle. This triangle is the **turtle**. It is the tool you will be using to draw. Notice the shape of the triangle. The narrow "point" shows the direction the turtle is facing.

In the Logo Window at the bottom of the screen you see a blinking vertical line. This is called the cursor. The flashing cursor tells you that Logo is waiting for you to give it instructions. Whatever you type appears at the location of the cursor. You can use your mouse and the standard I-beam cursor to move the location of the cursor to the particular place you want to insert text.

Drawing with the Turtle

Now you are ready to start drawing with the turtle. The turtle holds a pen that draws lines as the turtle moves. Try typing

```
FORWARD 50 (Enter)
```

or

```
FD 50 (Enter)
```

Be sure your press Enter and not Return. Also, be sure you type a zero (0), not the letter O ("oh").

Next type

RIGHT 90 (Enter)

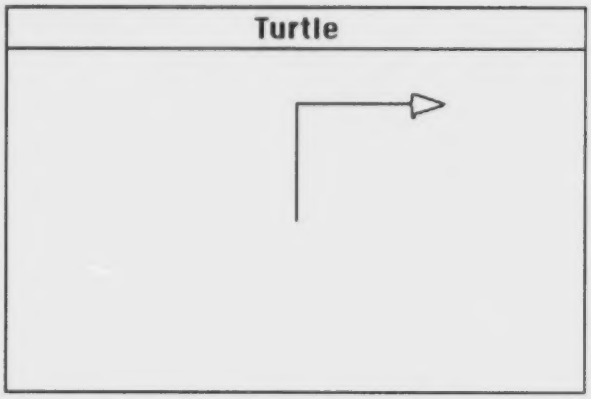
or

RT 90 (Enter)

Again, type

FORWARD 50 (Enter)

Does your Turtle Window look like this?



Can you finish drawing the square without typing any more commands?

Next type

CLEARGRAPHICS (Enter)

or

CG (Enter)

CLEARGRAPHICS clears the screen and moves the turtle to the center, facing up. You can also use

CLEAN (Enter)

This command clears the screen but does not move the turtle.

What happens if you type

FORWARD

and press Enter? You see

FORWARD NEEDS MORE INPUTS

Logo is telling you that you forgot the number after FORWARD. What happens if you misspell FORWARD?

```
FORWARD 50
THERE IS NO PROCEDURE NAMED FROWARD
```

What happens if you forget the space between FORWARD and 50?

```
FORWARD50
THERE IS NO PROCEDURE NAMED FORWARD50
```

Logo is telling you it doesn't understand your typing error. Just try again. Use your mouse to move the I-beam cursor to the error in the last line you typed. You can then use the arrow keys and the Delete key to correct your mistake.

Don't worry about making typing errors in Logo. Logo will tell you what it doesn't understand. You can then quickly remedy the problem. Logo usually gives you helpful messages when you make a mistake. Read them carefully.

Below is a list of some of the commands the turtle understands:

FORWARD *number*	BACK *number*
FD *number*	BK *number*
RIGHT *number*	LEFT *number*
RT *number*	LT *number*
PENUP	PENDOWN
PU	PD
CLEARGRAPHICS	
CG	

Experiment with these commands to see how each one works. Remember to press Enter after each command. After you have experimented for a while, decide on a design or drawing to make. If you make mistakes you can use CLEARGRAPHICS or CLEAN and start again. Remember that you can correct errors by using your mouse to move the I-beam cursor. Then you can edit the line and save yourself typing time.

If you create a drawing you want to keep, you can save it by using the SAVEPICT command. Put a data disk that can be used to store files in the disk drive, then select a meaningful name for your picture. For example, you might type

```
SAVEPICT "MY.PICTURE
```

Your picture is saved on disk. Notice that there is one quotation mark (") before the name of your picture file. Note also that there is no space after the quotation mark.

To see your picture again, type

```
CLEARGRAPHICS
LOADPICT "MY.PICTURE
```
(Don't forget the quotation mark before the name.)

Your picture appears on the screen.

Tips and Techniques

When using LOADPICT or SAVEPICT, you must always have a space and then a single quotation mark (") before the name of the picture. There is no space after the quotation mark. There is no second quotation mark after the name.

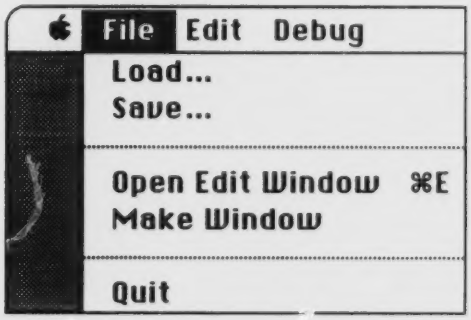
Important note: When you save using SAVEPICT, only the graphics are saved. None of the commands you typed are saved.

Spaces are very important in Logo. If you get error messages you don't understand, look carefully at what you typed. For example, you may not have any spaces in the name of a picture file. On the other hand, you must have spaces between a command like FORWARD and the number after it.

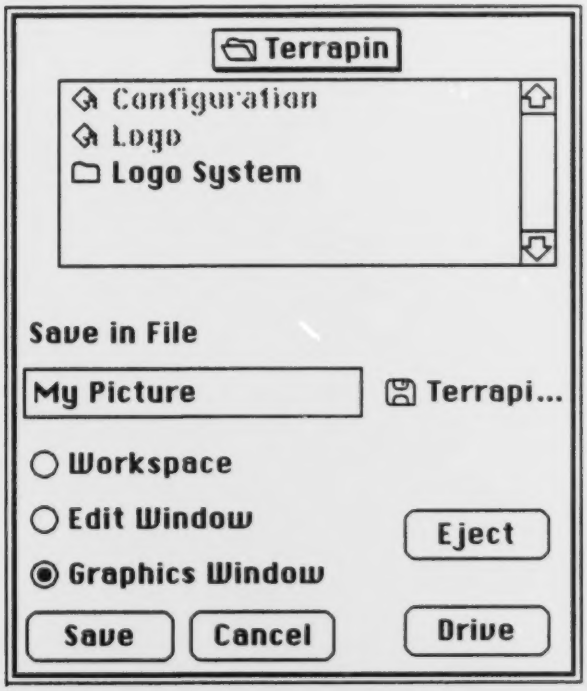
Beginners sometimes find the differences between the commands FORWARD and BACK and the commands RIGHT and LEFT difficult to understand. FORWARD and BACK are movement commands. They cause the turtle to take "turtle steps." RIGHT and LEFT are turn commands. The turtle doesn't move at all. Instead it turns about its middle, like a ballet dancer spinning on her toe.

It is not unusual to want to erase lines you draw by accident. It is possible to do this using PENERASE (or PE). You type PENERASE and draw over the line you don't want. Then you must type PENPAINT to continue to draw. Unless you erase a line immediately after drawing it, it can be very difficult to erase it completely. Generally it is better to start over rather than to try to fix several small errors on a drawing.

You can use the File Menu at the top of the screen to save and load pictures. If you click on File, you see



If you select the Save option, you see



You must be sure to click on the Radio Button next to the Graphics Window in order to save turtle pictures. You can then replace the automatic name "Turtle" with the name you wish to use for the file. Similarly, you can use the mouse and the pull-down menus to load a picture.

When you are finished working with Logo, always be sure you have saved any pictures you want to keep before you turn off the computer or let someone else use it. If you want to be sure your file is on disk, you can click on File on the Menu bar and then select either Save or Load. You then see a dialogue box like the one above. You can use the scroll bar to the right of the list of files to see what files you have on disk.

Project Suggestion

Make a turtle drawing of your choice. If you can't think of any other kind of design to draw, you might want to try having the turtle draw your initials. When you have a design you are satisfied with, save it by using SAVEPICT. Remember that you can use PU and PD.

2. Using REPEAT

You have learned how to make pictures with the turtle. You have also learned how to save pictures on your files disk. Now you can begin adding more Logo commands to your new vocabulary.

Using REPEAT

Clear the screen by typing

```
CLEARGRAPHICS
```

and then type

```
FORWARD 50
RIGHT 90
```

Remember to press Enter after each line. Remember that you can repeat lines by using the mouse to place the I-beam cursor. You can also use the up and down arrow keys to move the cursor to an insertion point or to a line you want to run. As long as the cursor is on a line, pressing Enter will cause Logo to run that line.

However, there is an easier way to have Logo repeat actions. Type

```
REPEAT 4 [FORWARD 50 RIGHT 90] (Enter)
```

What happens?

REPEAT is a new command. It must be followed by a number and a list of commands. Notice that the list of commands is contained in square brackets, [and]. Try something different. Type

```
CLEARGRAPHICS
REPEAT 90 [FORWARD 80 RIGHT 178]
```

Don't forget to press Enter. You get a "fluffy" ball. How about

```
CLEARGRAPHICS
REPEAT 18 [FORWARD 20 RIGHT 45 BACK 10 LEFT 25]
```

or

```
CLEARGRAPHICS
REPEAT 360 [FORWARD 1 RIGHT 1]
```

Take some time to experiment with REPEAT. You can get some fascinating patterns and designs. If a design you make would look better without the turtle showing, you can type

```
HIDETURTLE (Enter)
```

or

```
HT (Enter)
```

to hide the turtle. When you want to see the turtle again, you can type

```
SHOWTURTLE (Enter)
```

or

```
ST (Enter)
```

to see the turtle again.

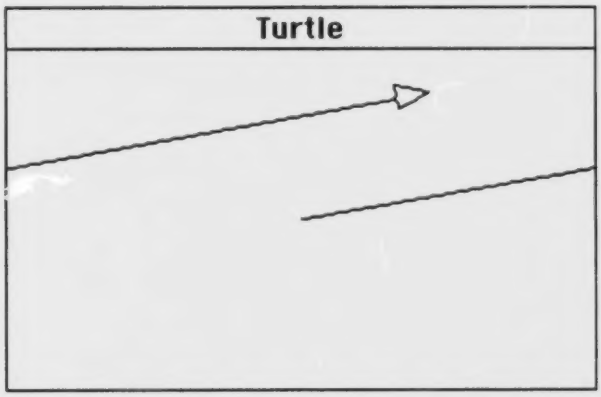
Don't forget that you can save any design you want to keep by using SAVEPICT followed by a name of your choice.

Wrapping

Have you discovered what happens when the turtle goes off the edge of the screen? If you type

```
CLEARGRAPHICS
RIGHT 80
FORWARD 300
```

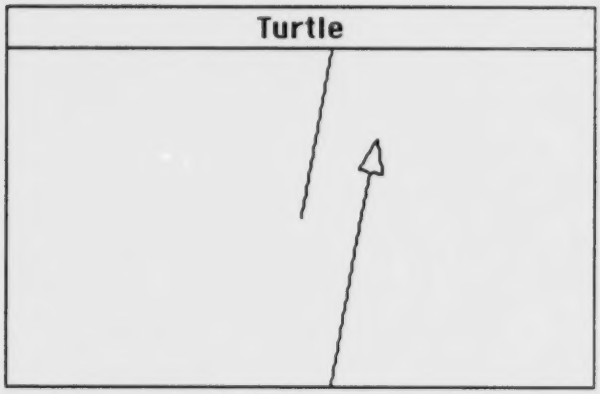
you will see that when the turtle goes off the screen on the right, it reappears again on the left at the same distance from the bottom of the screen. This is called "wrapping."



If you type

```
CLEARGRAPHICS
RIGHT 10
FORWARD 170
```

you see that it wraps around the top and bottom of the window as well:



Adding Text to a Drawing

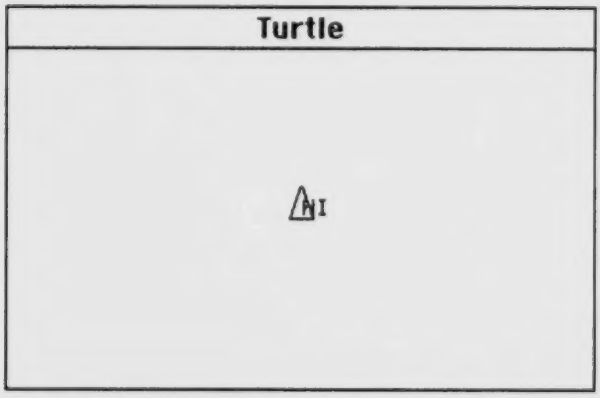
Be sure you have saved everything you want to keep. Then type

```
CLEARGRAPHICS
```

To put text on the screen, you use the STAMPTEXT command. STAMPTEXT puts a single word at the position of the turtle. It takes three inputs: a word, and the width and height for the window that contains the text. If you type

```
STAMPTEXT "HI 20 10
```

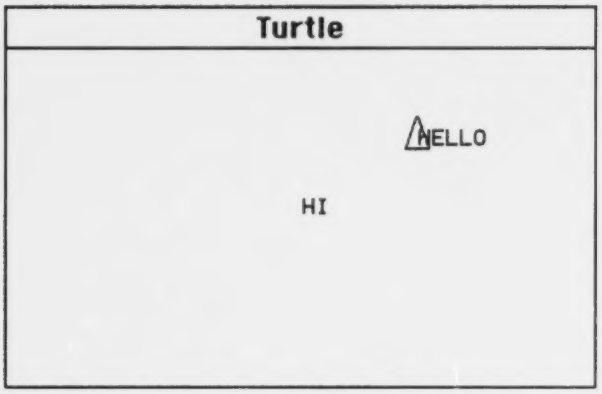
you see



You can also position the turtle using the SETXY command. SETXY moves the turtle to the coordinate specified. (You will learn more about SETXY in Chapter 36 of this book.) Thus, if you type

```
PU
SETXY 50 30
STAMPTEXT "HELLO 30 10
```

you see



Tips and Techniques

You must use *square brackets*, [and], with REPEAT, not parentheses, (and).

For now, you cannot use STAMPTEXT to put more than one word at a time on the screen. Later in this book (Chapter 7), you will learn how to use STAMPTEXT to put several words at a time at the location of the turtle.

Keep in mind that STAMPTEXT puts graphics characters on the screen. STAMPTEXT does not provide you with a word processor. You should not plan to use it to write long stories or letters since you have only the space in the Turtle Window available.

Project Suggestions

Now that you know about drawing with the turtle as well as using STAMPTEXT, create a drawing that is not in the center of the screen and label it with some text. When you are satisfied with your drawing, be sure to save it.

If you can't think of anything else to do, use REPEAT to make a flower in the upper left corner of the screen and then use STAMPTEXT to put a few words describing the flower in the lower right corner of the screen.

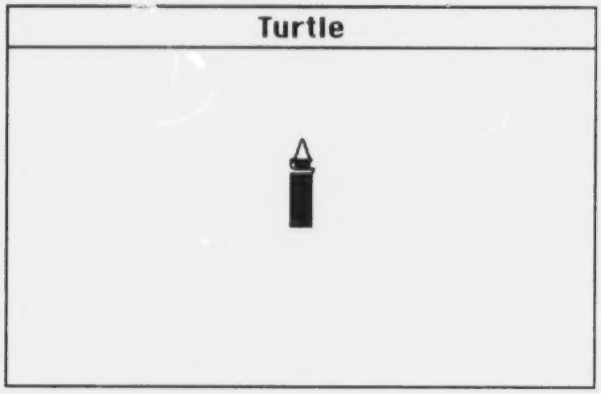
3. Changing the Pen and RANDOM

Changing the Pen

So far you have created drawings with the turtle that were single black lines on a white background. In this chapter you will learn that the pen the turtle carries can be changed to draw in different sizes and with different patterns.

You can change the size of the pen the turtle carries by using SETPENSIZE (SETPS). Try typing

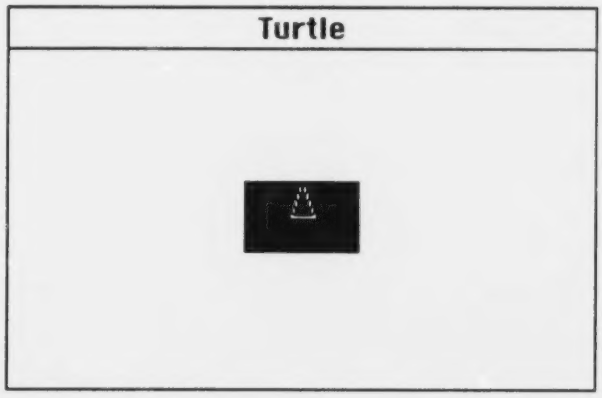
```
CLEARGRAPHICS
SETPENSIZE 10 10
FORWARD 20
```



The turtle draws with a square that is ten turtle steps on a side. Next, try typing

```
CLEARGRAPHICS
SETPENSIZE 50 30
FORWARD 1
```

You see

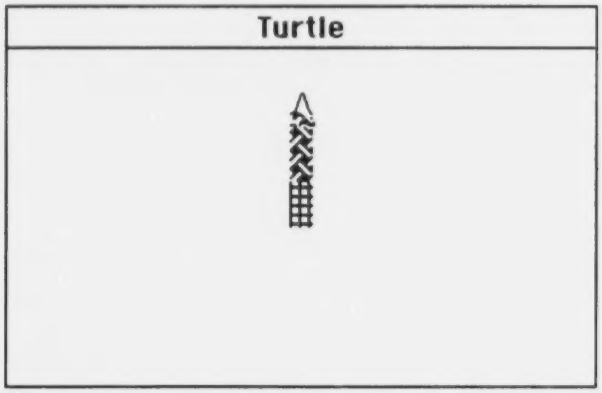


The turtle draws with a pen that is 50 turtle steps wide and 30 turtle steps high. Notice that the center of the pen is in the middle of the turtle. When you set the pen width quite large, the turtle will appear to be entirely within the line you are drawing. This can be confusing at first. Note that the pen can be set to draw with a rectangle anywhere from 0 to 255 on either side.

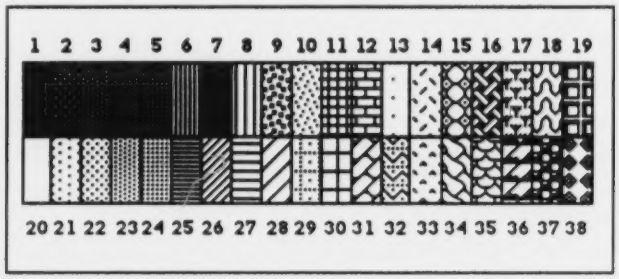
In addition to changing the pen size, you can change the pattern used by the pen with the SETPPATTERN (SETPP) command. Try typing

```
SETPENSIZE 20 20
SETPPATTERN 11
FORWARD 20
SETPPATTERN 20
FORWARD 20
```

You see



The pen can be set to any one of 38 patterns shown below.



Using RANDOM

Everything you have told the turtle to do up until this chapter has been exact: "go forward 50 steps," "turn left 57 degrees," or "draw with pen pattern 3." However, Logo also has the ability to create random numbers. Random numbers are numbers that have no apparent pattern to them. RANDOM is different from the Logo primitives you have seen so far. If you type

```
RANDOM 50
```

Logo responds with

```
RESULT: some.number
```

That is because RANDOM is an **operation**. Another name for a Logo operation is **reporter.** An operation or reporter returns a value, reporting it back to Logo. You must tell Logo what to do with the number. For example, try typing

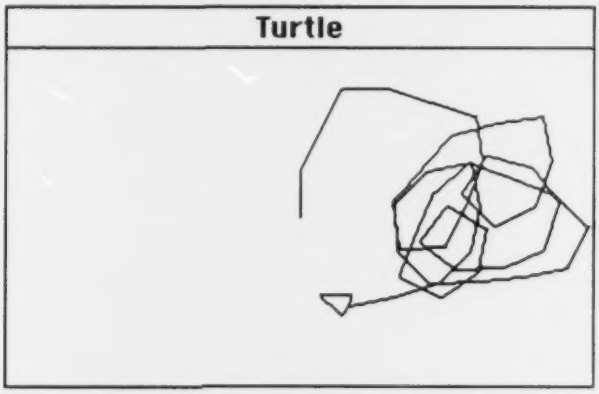
```
FORWARD RANDOM 50
```

Repeat this instruction a number of times. Remember, you can use the arrow keys or the mouse to move the cursor. Do you see that the turtle moves different amounts at different times? RANDOM reports to FORWARD a number from 0 to 49, and then the turtle moves as directed.

You can send the turtle on a random walk by typing something like

```
REPEAT 50 [FORWARD RANDOM 20 RIGHT RANDOM 90]
```

Does your design look anything like this?



Experiment with RANDOM. Can you get some other interesting effects?

How about picking random pen patterns? That certainly is possible. Recall that the patterns are numbered 1 to 38. So, if you use RANDOM 38, it will report a number between 0 and 37– almost what you want. You only need to add 1: 1 + RANDOM 38. You can get interesting effects by typing

```
REPEAT 10 [SETPPATTERN 1 + RANDOM 38 FORWARD 10 WAIT 15]
```

WAIT 15 causes Logo to pause for one-fourth of a second (WAIT 60 = 1 second). Or you can change the background color each time the turtle draws a line. Try this:

```
CLEARGRAPHICS
SETPENSIZE 10 10
REPEAREPEAT 20 [SETPPATTERN 1 + RANDOM 38 FORWARD 50 RIGHT 90 WAIT 10]
```

Try experimenting with random pen patterns. You can get some beautiful designs by simply changing the pen pattern randomly each time you move the turtle.

When you finish working with graphics, it is a good idea to reset the pen size and path to their original values:

```
CLEARGRAPHICS
SETPENSIZE 1 1
SETPPATTERN 1
```

Tips and Techniques

Using RANDOM is a bit like drawing numbers "out of a hat." You cannot predict what numbers will turn up next. If this is a new idea for you, experiment with the commands

```
CLEARGRAPHICS
FORWARD RANDOM 50
```

until you see that the turtle moves a different amount each time.

Keep in mind that you can often save yourself a lot of typing time by using the arrow keys or the mouse to repeat the last line you typed. Edit the line using either the arrow and delete keys or the mouse and I-beam cursor. Note that you do not have to be at the end of the line to press the Enter key to run the line.

As you work with RANDOM in the future, you may note that every time you turn off the computer and later turn it on and then type the same instructions, RANDOM produces identical results. You can avoid this happening by using the SETRANDOM command. SETRANDOM takes a number for input. This number "seeds" or starts the sequence of numbers generated by the random number generator. Unless you want to get the sazme sequence of numbers each time, you should use a different input to SETRANDOM each time you use it. One way to accomplish this is to use DATE. DATE reports a list containing the month, day, and year. The following instruction will seed the random number generator differently each time you use it:

```
SETRANDOM ((ITEM 1 DATE) + (ITEM 2 DATE) + (ITEM 3 DATE))
```

This will probably not be a problem, however, until you begin to write procedures.

Project Suggestions

Create a screen using different pen settings and RANDOM. You migh

- send the turtle on a random walk with a random pen pattern or width at each move.
- draw geometric figures using random patterns.
- fill the screen with randomly patterned designs.

When you are finished, be sure to save your picture using SAVEPICT. If you have a printer that works with *Terrapin Logo*, then you can use the PRINTWINDOW command to print your picture. For example, you might type

```
PRINTWINDOW "TURTLE
```

If your printer doesn't work with *Terrapin Logo*, then you can use an application such as MacPaint to print your picture.

4. Shapes and STAMP

Snapping a Turtle Shape

So far, the turtle shape has always been only a triangle. You can change the shape of the turtle to anything you can draw on the screen. First, you must draw a shape using the turtle. For example, you might type

```
REPEAT 4 [FORWARD 15 RIGHT 90]
```

This draws a small square on the screen. Next, use the SNAP command to "grab" a part of the screen. Type

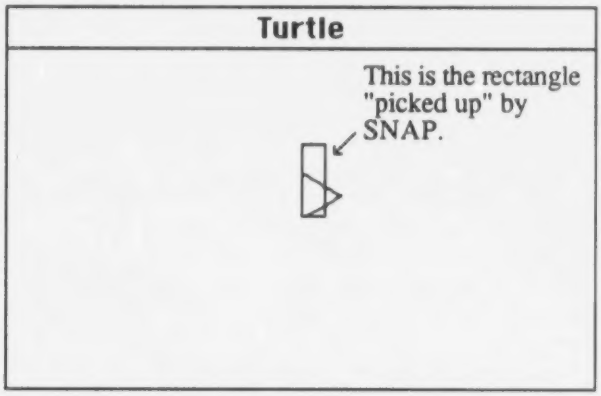
```
MAKE "MY.SQUARE SNAP 20 20
```

Now a 20 pixel (dot) by 20 pixel portion of the screen is stored with the name MY.SQUARE. This square includes the small square, which was 15 pixels on a side. The area "picked up" by SNAP has its lower left corner at the turtle's position.

Of course, the area you pick up need not be a square, for example,

```
REPEAT 3 [FORWARD 20 RIGHT 120]
MAKE "MY.TRI SNAP 10 30
```

In this case, a triangle 20 turtle steps on a side is drawn. Since the first input to SNAP was 10, only part of the triangle is SNAPped. Recall that the area "picked up" would start at the turtle position and include a rectangle 10 pixels wide and 30 pixels high.

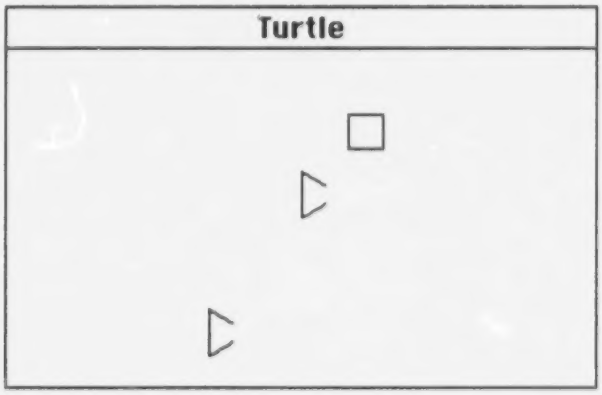


Stamping the Turtle Shape

Now you have two turtle shapes, a square, and a part of a triangle. You can place those shapes any place on the screen by using the STAMP command. Thus, you might type

```
SETSHAPE :MY.SQUARE
PU
SETXY 20 30
PD
STAMP :MY.SQUARE
PU
SETSHAPE :MY.TRI
SEXTY -40 (-60)
PD
STAMP
PU
HOME
```

You see



Notice that when you *use* the name of a shape, it is preceded by a colon (:), called "dots" in Logo. You will learn more about this symbol later in this book. Also note that the pen must be down when you stamp. If you leave the pen up, a copy of the turtle shape will not be put on the screen.

Note that you now have two copies of the triangular turtle shape. Which one is the real turtle? Type

```
FORWARD 20
```

and you will see that one of the turtles moves and the other stays put. This stamp of the turtle shape will remain on the screen until you type CLEARGRAPHICS.

Take note of the sequence of commands to get a copy of the turtle:

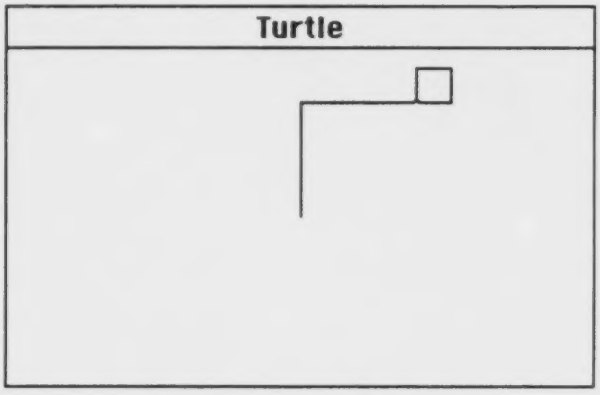
- Put the pen down with PD.
- STAMP the shape.
- Lift the pen (so there is no trail).
- Move the turtle (so you can see the stamped copy).

You can stamp any number of shapes in any number of places on the screen. Try it!

Drawing with Turtle Shapes

The turtle can draw while wearing a "costume." Type

```
CLEARGRAPHICS
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
```



Notice that the line doesn't seem to be coming out of the back of the shape in the same way as when the turtle was a triangle. Different shapes will touch the line at different places.

If you want to get back the turtle shape, you can type

```
SETSHAPE ASK 31 [SHAPE]
```

This command copies the turtle shape from another turtle (number 31). You will learn more about using other turtles later in this book.

Technically, SNAP returns a part of the screen that is one pixel wider and one pixel higher than the numbers you give it. This is a technical matter that should not concern you unless you are doing very complex or detailed work.

Tips and Techniques

If you stamp a turtle on top of an already stamped turtle shape, the turtle will disappear. If you move the turtle on top of a stamped turtle shape, both may disappear. Don't panic. Just move the turtle.

You can erase a stamped turtle shape by moving the turtle (costumed in the same shape) on top of the shape you don't want and then stamping. This process can be confusing. When the turtle shape is exactly over the shape you want to erase, both the turtle and the shape disappear. Just type

```
PD
STAMP
```

The turtle will reappear and your stamped shape will be gone.

Project Suggestions

By combining designs made with the turtle, stamped shapes, and STAMPTEXT, you can create a variety of interesting screens. For example, you might create a

- postcard with a picture and a greeting
- party invitation
- sign that advertises the next school dance
- cover for a report
- campaign poster.

5. Defining New Shapes

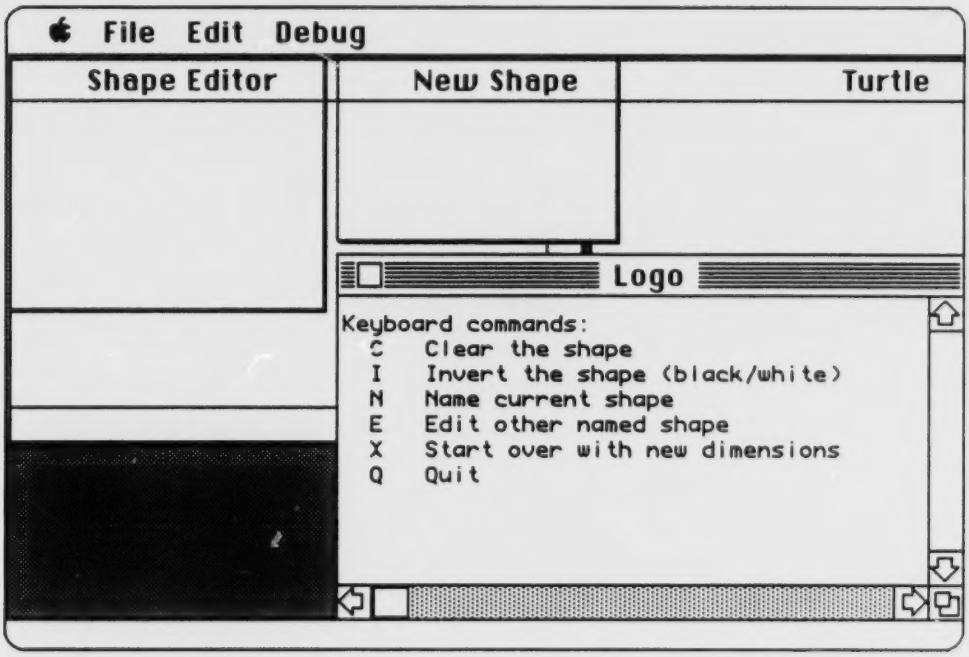
In the last section, you learned how to create new turtle shapes using SNAP. You can also create new turtle shapes using the Shape Editor. To begin, put your disk containing the *Terrapin Logo* language in the disk drive, load the Shape Edit file from the disk, and type

```
SHAPE.EDIT
```

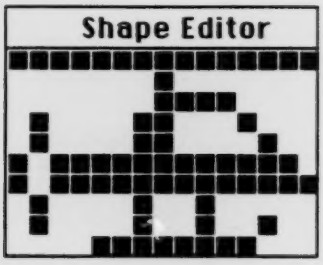
You will be asked how large a shape you want. For example, suppose you want to create a helicopter shape. You begin by typing the dimensions you want. For the helicopter, you might enter

```
15 10
```

You then see several windows appear on the screen.

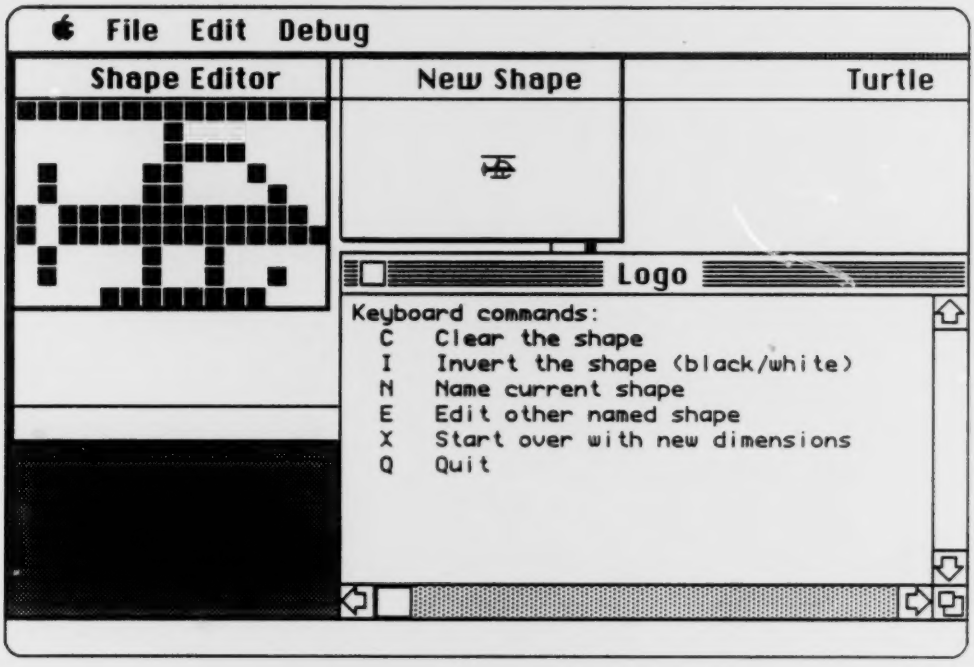


The Shape Editor Window works much like Fat Bits in MacPaint. Use your mouse to click on the dots you want to use to form your helicopter. For example, your shape might look like this:



In the Logo Window, you see commands to use in the Shape Editor. When you are finished with your shape, type "N" to name your shape. You are then asked if you want a special pen location. This allows you to determine where the "ink" flows from the shape. You can simply answer "N" to this question for now.

Next, you are asked for a name for the shape. Type the name you want. Your shape will then be defined. Finally, you are asked if you want to see the shape in normal dimensions. You can answer "Y" to that question for now.



This brings you back to the Shape Editor menu. You can then define another shape or leave the Shape Editor. When you are finished defining shapes and you leave the Shape Editor, *you must close the Shape Editor windows* or the turtle will not appear in the Turtle Window.

To use your new shape, you type

```
SETSHAPE :HELICOPTER
```

and the turtle changes to the shape of a helicopter. You can now stamp your new shape on the screen or use it to draw.

In general, to use a shape you have defined, you type

```
SETSHAPE :name.of.your.shape
```

Be sure you put the colon symbol (:), called "dots", in front of the name.

To save shapes that you create, you must first load the Saveshapes file from disk. Decide what shapes you want to save. You must list the names of all the shapes you want to save and decide on a file name for your shapes. To save only your helicopter shape, you type

```
SAVESHAPES [HELICOPTER] "HELI
```

This will save your shape in a file called HELI. If you have more than one shape, you must list all the shape names. In general, you use

```
SAVESHAPES [list of shape names] "file.name
```

Note that there is a quotation mark before the name of the file. The shapes you list will be saved using the file name you choose. To reload the shape file containing your helicopter shape, type

```
LOAD "HELI
```

In general, you load a file of shapes by typing

```
LOAD "file.name
```

You now refer to your shapes by name.

Note: Both the Shape Edit and the Saveshapes files load procedures into Logo's memory. When you have saved your shapes in a file, type

```
ERALL
```

to clear the memory. This removes all the procedures *and* the shapes from memory. You will want to be particularly careful to remove all the extra procedures from memory when you are writing your own procedures. Otherwise, you will find that all these extra procedures will become part of your workspace. Always create your shapes file, clear the memory, and then work on your program by reloading the shapes file you need.

If you want to make a copy of a shape you have defined using the Shape Editor, you must first be sure that you have loaded Shape Edit and that you have loaded your shape file into memory. Then type

```
SHAPE.EDIT
```

Enter the width and height of the shape you want to edit. Next, enter the name of the shape to edit. It will appear in the New Shape Window and the Shape Editor Window. Now, give the shape a new name. You now have two copies of a shape with different names. Note that this is an easy way to make small modifications to a shape. Make several copies, change each one slightly, and finally, give each a new name.

Tips and Techniques

Using the Shape Editor program in *Terrapin Logo* takes some practice. Here are some things to keep in mind to help you avoid trouble.

1. You must load the Shape Edit program in order to use create new turtle shapes.
2. You must type SHAPE.EDIT in order to use the Shape Editor.
3. Follow the directions carefully when you are working in the Shape Editor.
4. Be sure to close the Shape Editor windows when you are ready to use a newly defined turtle shape.
5. Occasionally strange things happen to the text in the Logo Window. Don't panic–it still works!
6. You must load the Saveshapes file in order to save any shapes you create.
7. Use the SAVESHAPES command to save your shapes in a file name of your choice.
8. Always remember to use ERALL to clear the procedures from the Shape Edit and Saveshapes files from memory.
9. In order to use any shapes, you must load the file of shapes into memory.
10. To use a particular shape, you use the SETSHAPE command followed by the name of the shape. Be sure to use : (dots) before the name of the shape.

Project Suggestions

Now that you can create any shapes you want, try one of the projects from the previous chapter, but use your own shapes. For example, you can make

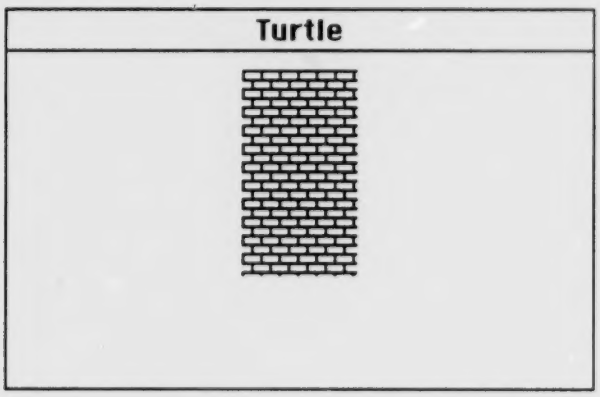
- more personalized greeting cards
- signs with special symbols on them
- cartoons.

6. Using FILLSH

You now know how to change the pen size and the pen pattern. Using this combination of commands, you can create rectangular areas on the screen that are filled with the pattern of your choice. For example,

```
CLEARGRAPHICS
SETPPATTERN 12
SETPENSIZE 50 50
FORWARD 40
HT
```

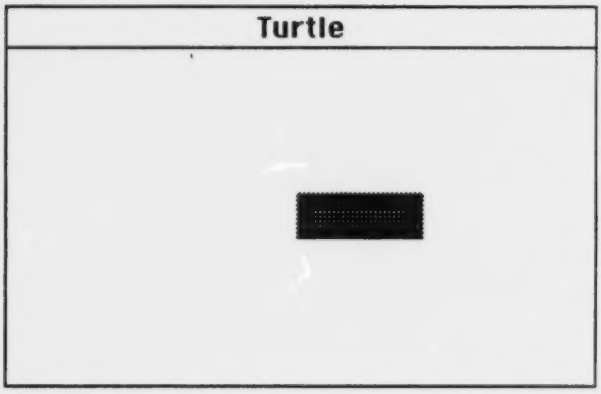
gives you



and

```
CLEARGRAPHICS
SETPPATTERN 5
SETPENSIZE 5 20
RIGHT 90
FORWARD 50
HT
```

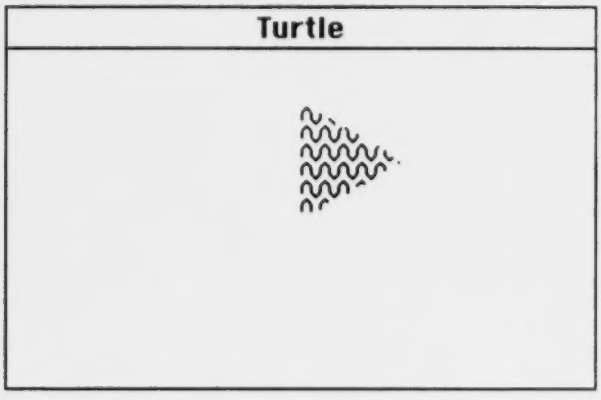
gives



However, there are times when you may want to fill a shape that is not rectangular. To accomplish this, you can use the FILLSH command. FILLSH takes a list of Logo commands as input. Thus, if you wanted to fill a triangle, you might type

```
CLEARGRAPHICS
SETPENSIZE 1 1
SETPPATTERN 18
FILLSH [REPEAT 3 [FORWARD 50 RIGHT 120]]
```

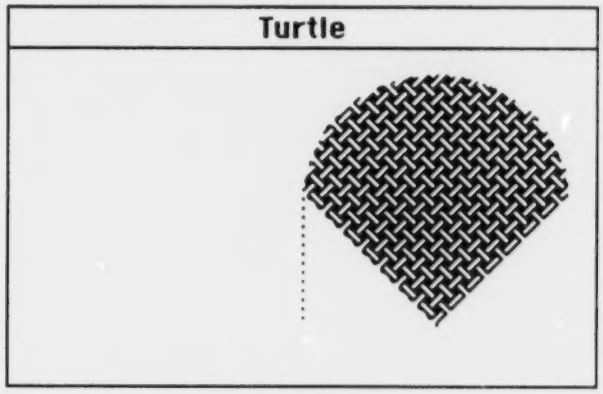
You see



You can fill any shape you wish this way, for example,

```
CLEARGRAPHICS
SETPENSIZE 1 1
SETPPATTERN 16
FILLSH [REPEAT 18 [FORWARD 10 RIGHT 10] RIGHT 45 FORWARD 80 RIGHT 90
      FORWARD 80]
```

When you type the above instructions, you see



Note that you can fill any area with the pattern of your choice. However, you may get unexpected results, like the extra line in the above design.

Tips and Techniques

Since filling areas makes major changes on your screen, it is a good idea to save your work before you use FILLSH. Suppose you save your picture by typing

```
SAVEPICT "MY.PIC
```

Next you fill an area. The FILLSH doesn't work right, and you probably want to begin again. First, type

```
LOADPICT "MY.PIC
```

and try again.

The use of brackets after FILLSH can be confusing. FILLSH needs a set of brackets to enclose all the commands that follow it:

```
FILLSH [commands.go.here]
```

If, for example, you have a REPEAT instruction after FILLSH, then you see something like

```
FILLSH [REPEAT 4 [FORWARD 50 RIGHT 90]]
```

At first glance it may appear that there are two right brackets in this instruction. In fact, there is a *set* of brackets for the FILLSH command and a *set* for the REPEAT command.

Project Suggestions

Create a number of closed shapes on the screen, each filled with a different pen pattern. Can you create some interesting artistic effects using filled areas?

Design a sign announcing some school activity. Use a combination of text and filling to make your sign.

Use the Logo techniques you have learned so far to create the cover for a report. You could create a turtle shape appropriate to the subject of the report. Remember, you can fill areas with pen patterns as well as stamp turtle shapes.

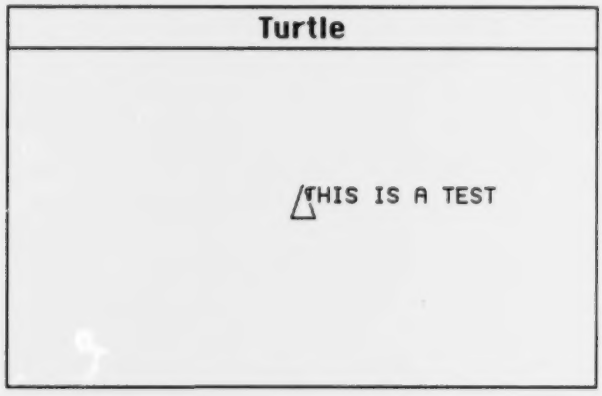
7. Putting Text on the Screen

You have seen that you can put text in the Turtle Window by using the STAMPTEXT command. However, so far you have learned that STAMPTEXT accepts only a single word as input. If you want to put more than one word on the screen, then you have to stamp a word, move the turtle, stamp the second word, and so forth. *Terrapin Logo* provides a way to put text on the screen by using *strings*. Strings are not usually found in Logo, so if you are familiar with a different version of Logo, this will be a new idea.

Begin by typing

```
STAMPTEXT `THIS IS A TEST` 100 15
```

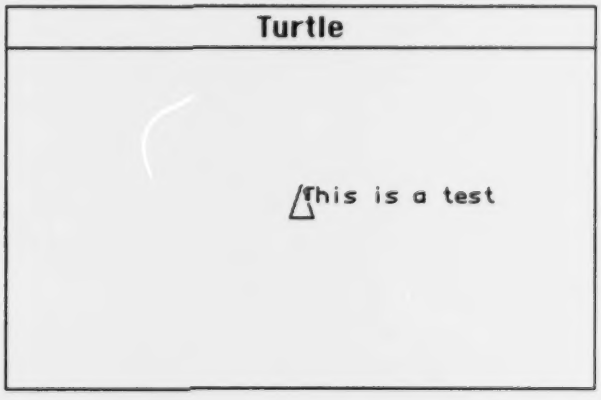
Be careful that you use the "back quote" symbol, not the usual single quotation mark. You see



Next type

```
STAMPTEXT `This is a test` 100 15
```

You see

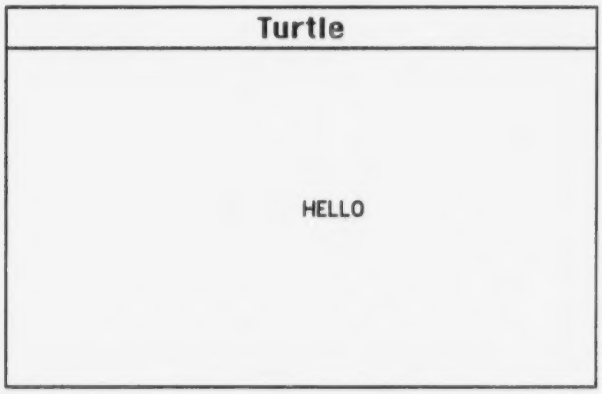


Using strings allows you to include both upper- and lowercase text on your screen.

In addition to putting "ordinary" text on the screen, you can display text using a variety of fonts and styles. The SETTFONT command takes the name of a font currently installed in the System Folder of your Macintosh. Thus, if you type

```
SETTFONT "GENEVA
STAMPTEXT "HELLO 50 10
```

you see



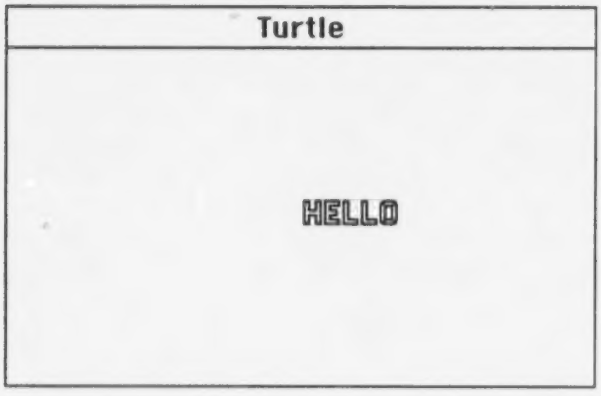
The style of the text used can also be changed with the SETTSTYLE command. This command allows you to change both the text style and the point size. The numbers corresponding to various styles are given below:

Number	*Style*
0	Normal
1	Bold
2	Italic
4	Underline
8	Outline
16	Shadow
32	Condensed
64	Expanded

To combine the features, you add the style numbers. The size input to SETTSTYLE is the point size. (Setting the point size to 0 causes Logo to use the default Macintosh size, 12.) Thus, you might type

```
SETTFONT "CHICAGO
SETTSTYLE [8 12]
STAMPTEXT "HELLO 50 10
```

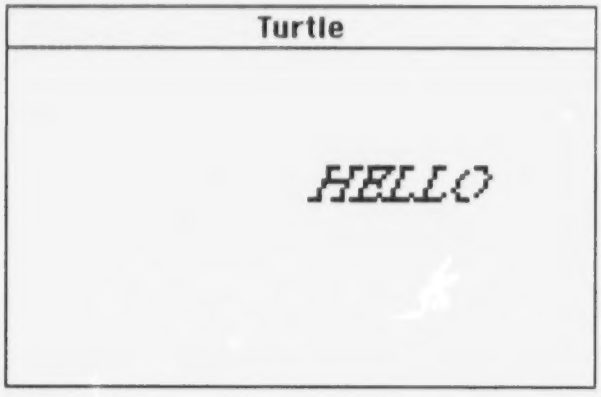
to see the word "HELLO" in 12-point Chicago outline:



Or you might type

```
SETTFONT "GENEVA
SETTSTYLE [2 24]
STAMPTEXT "HELLO 50 24
```

to see 24-point italic Geneva:



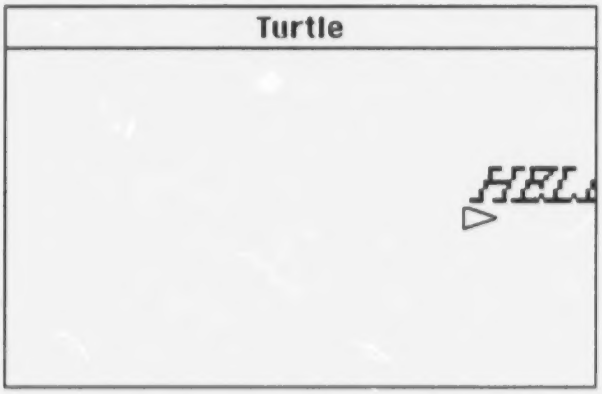
After using different fonts, you will probably want to reset the text to Logo's normal style, which is 9-point Monaco. Do this by typing

```
SETTFONT "MONACO
SETTSTYLE [0 9]
```

Tips and Techniques

The use of many different styles and fonts allows a great deal of variety in your screens. However, you should resist the temptation to overuse different fonts and styles to the point that your screens are cluttered.

If the text you are putting on the screen reaches the edge, it simply disappears. It does not wrap the way turtle drawings do.



If you are experimenting with different fonts and styles, be sure to reset the text to normal before turning over your computer to someone else. It can be very frustrating to find yourself with a strange font and no idea of how to fix the problem!

If you prefer not to use *Terrapin Logo*'s strings, you can use the more common words and lists to put text on the screen by creating a couple of tool procedures. (You will learn about writing procedures in Chapter 9 of this book.)

First, press Command-E (⌘-E) to go to the Logo Editor. A new window appears in the upper left corner of your screen. Type the following procedures in this window, pressing return at the end of each instruction line:

```
TO MAKE.WORD :LIST
OUTPUT MAKE.WORD.LOOP :LIST CHAR 32
END

TO MAKE.WORD.LOOP :LIST :NEW.WORD
IF EMPTY? :LIST [OUTPUT :NEW.WORD]
OUTPUT MAKE.WORD.LOOP BUTFIRST :LIST ( WORD :NEW.WORD FIRST :LIST CHAR 32 )
END
```

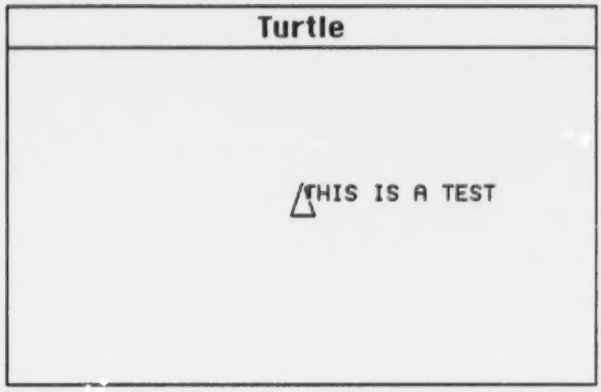
You must be very careful to include all the punctuation and spaces as shown. (You will learn all about such matters later in this book.) When you are finished, press the Enter key. In the Logo Window, you see

```
MAKE.WORD DEFINED
MAKE.WORD.LOOP DEFINED
```

Now you can use the "MAKE.WORD" tool to stamp more than one word on the screen at a time. For example, if you type

```
CLEARGRAPHICS
STAMPTEXT MAKE.WORD [THIS IS A TEST] 100 15
```

you see



Notice that you must put square brackets around the list of words you want printed. These are the same symbols you use with REPEAT.

If you use the MAKE.WORD procedures given above, be sure you carefully save them in a separate file. Type

```
SAVE "MAKE.WORD.PROCS
```

If you save your picture using SAVEPICT, the procedures will not be saved. You will need to load them again if you restart Logo at a later time. To load the procedures, type

```
LOAD "MAKE.WORD.PROCS
```

and the procedures will be loaded into memory.

Project Suggestions

Now that you have learned more about placing text on the screen, you can use this capability to do a variety of things, such as

- making a sign with a variety of text fonts and styles.
- using a variety of text styles to more expressively display a poem.
- making a design or picture using letters, numbers, and other keyboard symbols. (Pictures such as this used to be called "typewriter pictures.")

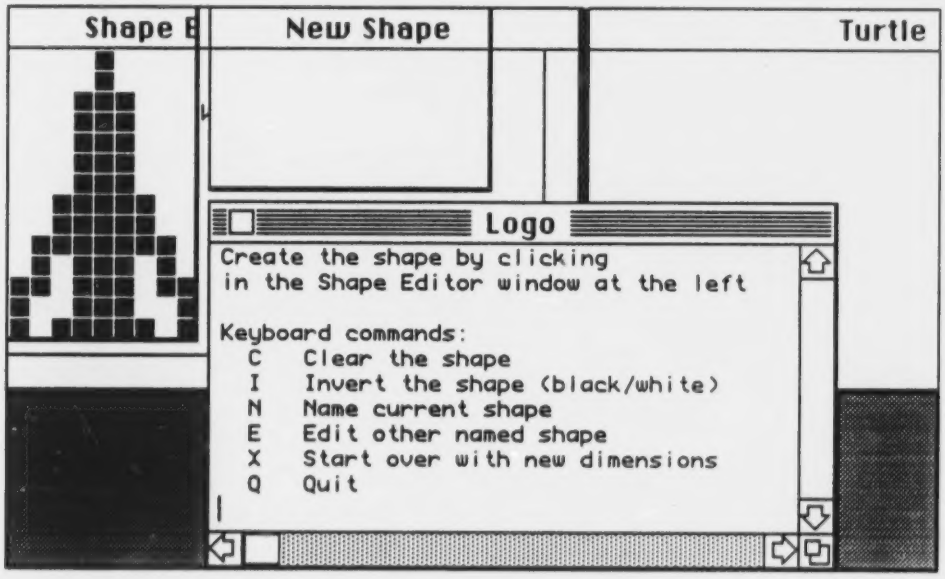
8. Mixing Text and Graphics

You have now learned how to put text on the screen. Earlier you learned how to put graphics on the screen using the turtle. You can combine what you have learned about using the turtle and working with shapes with what you have learned about using STAMPTEXT.

Begin by loading the Shape Edit file from the file menu. Then type

```
SHAPE.EDIT
```

to create a new shape. Make a rocket shape as shown below in a grid 9 pixels wide and 14 pixels high. Click on the center bottom pixel when asked about a special pen position so that the "ink" will flow from the back of the rocket.

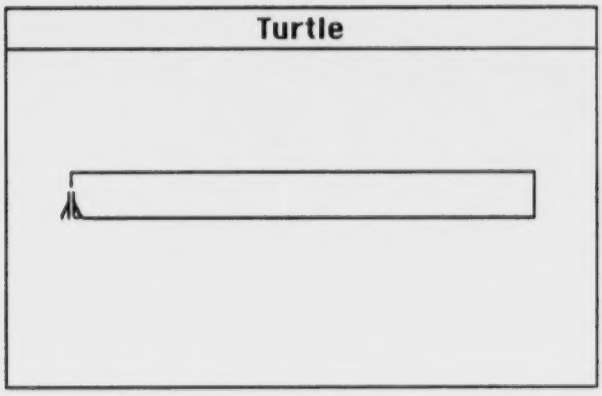


Name your shape ROCKET. Remember to load the Saveshapes file and save your rocket shape if you want to keep it for future projects. Close the shape Edit windows.

Now move your rocket-shaped turtle to the left side of the screen and draw a rectangle:

```
SETSHAPE :ROCKET
SHOWTURTLE
PENUP
LEFT 90
FORWARD 100
RIGHT 90
PENDOWN
REPEAT 2 [FORWARD 20 RIGHT 90 FORWARD 200 RIGHT 90]
```

Your screen looks like this:



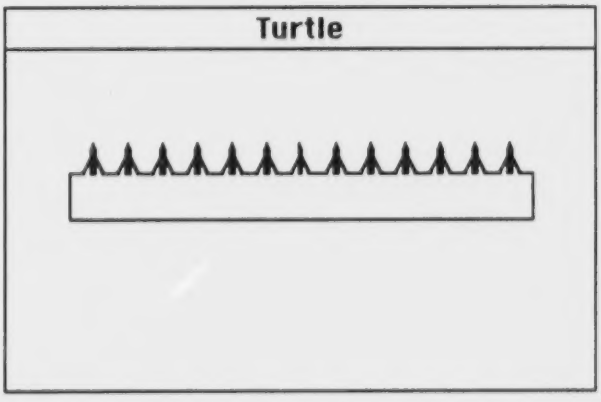
Now, move the rocket to the top of the rectangle:

```
PENUP
FORWARD 20
RIGHT 90
FORWARD 10
```

Then, stamp several copies of the rocket:

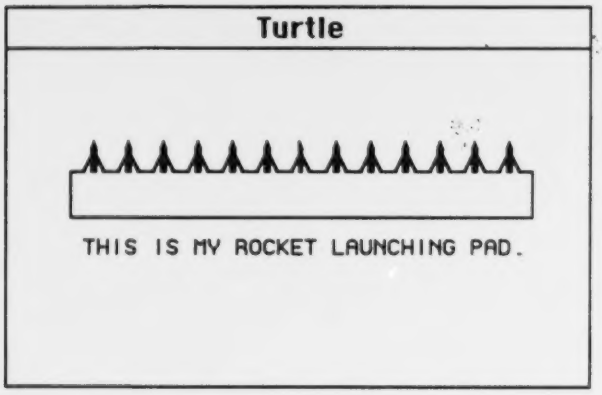
```
REPEAT 13 [PENDOWN STAMP :ROCKET PENUP FORWARD 15]
HIDETURTLE
```

Now you see



Place the turtle and add some text:

```
SHOWTURTLE
PENUP
LEFT 90
BACK 50
LEFT 90
FORWARD 205
STAMPTEXT `THIS IS MY ROCKET LAUNCHING PAD` 200 10
```



If you have the appropriate printer, you can print your screen using PRINTWINDOW, or you can save your picture by typing

```
SAVEPICT "LAUNCH.PAD
```

Tips and Techniques

Recall that when you use the Shape Editor, you load procedures into memory. It is a good idea to remove those procedures when you are finished by typing

```
ERASE PROCEDURES
```

Also remember to close all the extra windows created by the Shape Editor before you begin to use your new shape.

If you choose to use the MAKE.WORD procedures to stamp text on the screen, remember that they too are put into memory when you read them. You should also erase these procedures when you are finished by using ERASE PROCEDURES. You will learn more about working with procedures in the next chapter.

When you have put a lot of work into a Logo project, it is very frustrating to make a major error or accidentally erase your work. When you make a mistake, *don't panic*. Before you press any keys, take a moment to think.

Have you recently saved the picture? If so, then reload it with

```
LOADPICT "picture name
```

Have you been saving your pictures on a regular basis, using names like MY.PICT1, MY.PICT2, MY.PICT3, etc.? If so, you can use the File menu to examine the files on your current disk. Then you can read the most recent copy of your picture.

If you can't rescue your work this time, in the future you should make a habit of saving your work every 10 to 15 minutes. Better yet, save it every time you complete a major change on your screen. The time taken now to save your work will pay off later if you make a mistake.

Project Suggestions

Use a mixture of text and graphics to create screens of your choice. You might want to make a greeting card, a sign, or a cover for a report. Remember that you can create any shape you want by using the Shape Editor. Try extending some of the suggestions made at the end of Chapter 7 by adding graphics to them.

9. Writing Procedures

All the graphics you have created so far have been created by typing commands to put either designs or words directly on the screen. When you saved, you saved exactly what was created in the Turtle Window.

Have you ever created a neat graphics design that you would like to use in another project? Perhaps you remembered the commands you used to make it, perhaps not. Without a list of the commands, it can be hard to recreate a graphics design. The commands you see in the Logo Window are not saved. If you turn off your computer and start Logo at another time, you must begin again. If you used SAVEPICT to save the Turtle (or Graphics) Window, the commands you used to create it are not saved. Even if you use the pull-down File menu, you cannot save the Logo Window. Your only choices are to save the Workspace, Edit, or Graphics windows.

However, there is a way to save commands that you use to create pictures in Logo. To accomplish this, you use a part of the computer's memory called the *Editor*.

Recall that you can type

```
REPEAT 4 [FORWARD 50 RIGHT 90]
```

in the Logo Window and the turtle draws a square in the Turtle Window. When you type commands in the Logo Window, you are working in the *immediate mode*. You can type the same kind of commands using the Editor. Start by typing

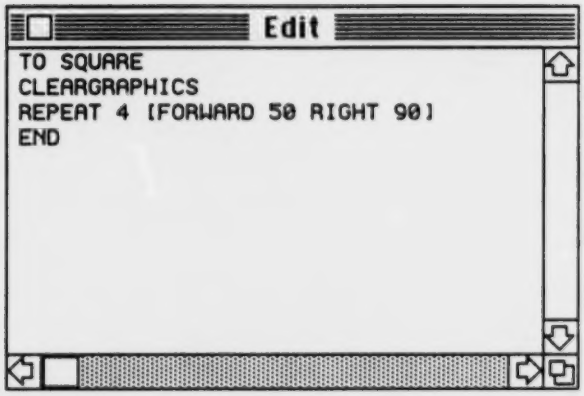
```
EDIT []
```

or

```
ED []
```

or press Command-E (⌘-E). You see

Next, enter the following lines in the Editor Window, pressing Return after each line:



When you are finished, press *Enter* to leave the Editor and define the procedure. You see

```
SQUARE DEFINED
```

Next, type

```
SQUARE
```

What happens? You have just written your first Logo procedure. A procedure is a collection of Logo commands. Procedures must begin with the word TO and end with the word END. The structure of a procedure is as follows:

```
TO name.of.procedure <–title line

 Commands go here <–body of the procedure

END
```

Procedures must be written in the Editor. Then you can save them. For example, to save your square procedure, you can type

```
SAVE "MY.SQUARE
```

Or, you can use the pull-down File menu, click the radio button next to Workspace, and type the name of the file.

The save command saves any procedures that are in the computer's memory (called the workspace). When you come back later and want to use your square procedure, you start Logo and then type

```
LOAD "MY.SQUARE
```

or you can again use the File menu. Select Load, be sure the Workspace radio button is checked, and select your file from the list of files. You then see

```
SQUARE DEFINED
```

That means that the square procedure is ready to be used.

Once you begin saving procedures, you will probably want to be able to print them on your printer. *Terrapin Logo* saves standard ASCII or text files. You can simply load those files into your favorite word processor and print them in whatever format or font you wish.

There is a good bit more to learn about the Editor and working with procedures. Procedures are kept in the part of the computer's memory called the *workspace*. You need to learn how to edit several procedures from your workspace and to remove procedures from your workspace. These topics will be addressed in the next few sections.

Although learning to write procedures is relatively simple, the idea embodied in procedure writing is profound. When you create a series of instructions to complete a particular task and then put them into a named procedure, you have "taught" Logo a new word. This new word behaves exactly as if it were a primitive. If you don't look in the Editor, you have no way of knowing whether the commands you type in immediate mode consist of primitives or user-defined procedures. The fact that one easily creates procedures that function exactly like primitives is what makes Logo an easily *extensible* language.

The Editor can contain as many procedures as the memory of the computer and/or the version of Logo you are using allows. Each of these procedures must have a unique name. Each new name becomes part of the Logo vocabulary available when you use that file.

Since all the commands you have learned so far can be used in a procedure as easily as they can be used in the immediate mode, you now have the tools to write procedures to do many interesting things.

You can move the turtle around:

```
TO PLACE.TURTLE
CLEARGRAPHICS
PENUP
FORWARD 20
RIGHT 60
FORWARD 30
PENDOWN
END
```

You can change the shape of the turtle and pen used (be sure the rocket shape is loaded):

```
TO CHANGE.TURTLE
SHOWTURTLE
SETSHAPE :ROCKET
SETPENSIZE 5 5
SETPPATTERN 18
FORWARD 50
END
```

You can stamp turtle shapes:

```
TO STAMP.IT
SETSHAPE :ROCKET
STAMP :ROCKET
PENUP
FORWARD 20
END
```

Take some time now to experiment with writing a procedure. Notice as you experiment how powerful this idea of procedure writing is. Now *you* are teaching Logo how to do new things. The computer is now under your control.

Tips and Techniques

When you begin using the Editor to write procedures, there are a number of things you should remember.

- When you are in the Edit Window, Return ends a line but Enter leaves the Editor and defines the procedures.
- Look for the cursor when you begin to type. Be sure you are typing in the Editor Window if you want to define procedures or in the Logo Window if you want to work in the immediate mode.
- When you define a procedure, it becomes a part of your workspace even though it is no longer in the Edit Window. You will learn more about the workspace in a later chapter.
- You can put a particular procedure back into the Edit Window by typing

```
EDIT [name]
```

or you can put several procedures into the Edit Window by typing

```
EDIT [name1 name2 name3]
```

Project Suggestions

Write a procedure to make a small design. In immediate mode, move the turtle to various places on the screen and draw your design using your procedure. Can you create a neat artistic effect?

Write a procedure to stamp a shape in the Turtle Window. In immediate mode, use a combination of REPEAT, FORWARD, RIGHT, and your procedure to make a border of shapes around the edge of the window.

Write a procedure to make an elaborate graphics design using the turtle. In immediate mode, use your procedure to make the design, and then use STAMPTEXT to add a description of your design on the screen.

Create a shape of your own and then write a procedure to stamp your shape in a pattern in the Turtle Window. Add some text using STAMPTEXT.

Write a procedure that draws a graphics design and changes the pen pattern randomly with each line it draws. Use STAMPTEXT to put a title on your design.

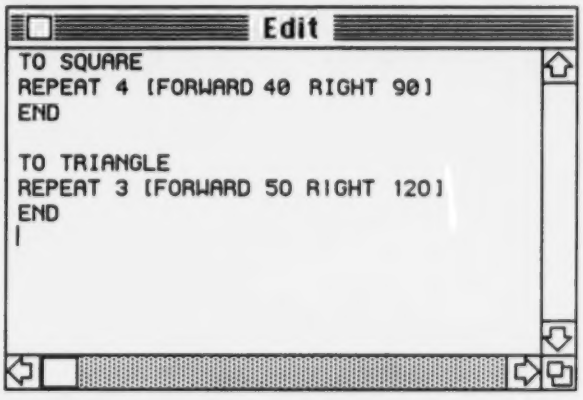
10. More Than One Procedure

You now know how to write a procedure using the Editor Window. As mentioned in the last section, you can write as many procedures as you want in the Editor as long as there is space.

The example given below shows you how to write a number of procedures and put them together into a program. As you work through it, remember that you use the Return key to end a line of a procedure and the Enter key to leave the Editor. You can use the arrow keys, Delete key, or the mouse to edit your work.

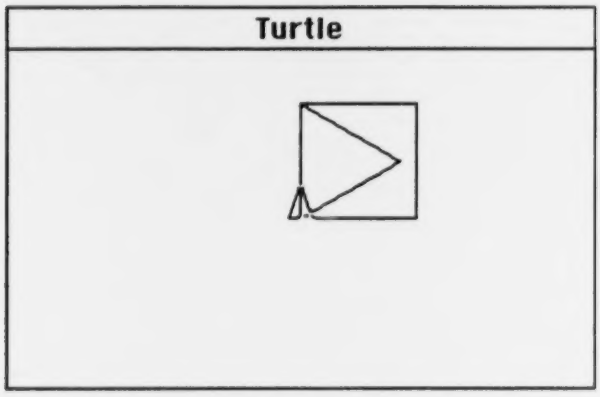
Suppose you want to have the turtle draw a house in the Turtle Window and then put a "For Sale" sign on it. You can, of course, do this in immediate mode. If, however, you do it with procedures, then you can easily make a number of identical houses.

To begin, notice that you can draw a simple house made up of a square for the main part of the house and a triangle for the roof. Go to the Editor Window and write these procedures for drawing a square and a triangle:



Press Enter to leave the Editor and define the procedures. Type

```
SQUARE
TRIANGLE
```



You see the parts of the house, but they are not in the right place. Type CLEARGRAPHICS and then see if you can figure out the necessary commands to get the roof in the right place. These commands should become part of a procedure to get to the roof. One solution might be

```
TO GET.TO.ROOF
FORWARD 50
RIGHT 30
END
```

Use Command-E (⌘-E) to go to the editor. This will leave the SQUARE and TRIANGLE procedures still in place. Add the GET.TO.ROOF procedure. Leave the Editor by pressing Enter and type

```
CLEARGRAPHICS
SQUARE
GET.TO.ROOF
TRIANGLE
```

Oops! The house is off the top of the screen! Add yet another procedure to the Editor:

```
TO SET.UP
CLEARGRAPHICS
PENUP
BACK 40
PENDOWN
END
```

Now type

```
SET.UP
SQUARE
GET.TO.ROOF
TRIANGLE
```

Does the house look the way you want it to? These procedures can become part of a HOUSE procedure:

```
TO HOUSE
SET.UP
SQUARE
GET.TO.ROOF
TRIANGLE
END
```

Now, you can type HOUSE in the immediate mode and see your house.

Next, you need to work on the "For Sale" sign. You can use the STAMPTEXT command to put the text on the screen. Perhaps you will also want to set the font and text style as well. Thus you might add this procedure in the Editor:

```
TO MAKE.SIGN
PENUP
RIGHT 60
FORWARD 55
LEFT 90
BACK 25
SETTFONT "MONACO
SETTSTYLE [8 9]
STAMPTEXT `FOR SALE!` 70 15
HIDETURTLE
END
```

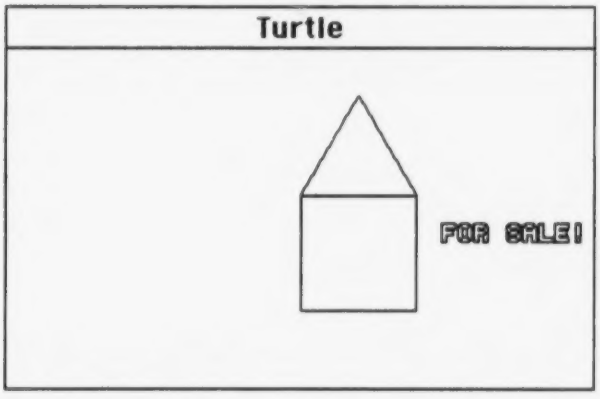
Now you can combine your HOUSE procedure and your MAKE.SIGN procedure:

```
HOUSE
MAKE.SIGN
```

The house appears with the sign next to it. Now you can put this all together in one procedure. Go to the Editor Window and type

```
TO HOUSE.AND.SIGN
HOUSE
MAKE.SIGN
END
```

Now, type HOUSE.AND.SIGN in the immediate mode and you see

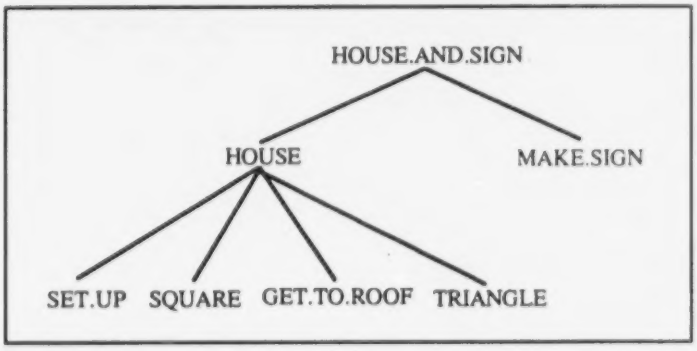


When you are writing a program with a number of procedures, it is a good idea to include a description of the program. In the Editor, move the cursor to the HOUSE procedure. Any text that follows a semicolon (;) is a comment. Thus, you might change HOUSE as follows:

```
TO HOUSE.AND.SIGN
;This program draws a house with a
;"For Sale"sign.
HOUSE
MAKE.SIGN
END
```

The line following the semicolon (;) is ignored by Logo. The procedure runs exactly the same with or without the comment.

The procedure HOUSE.AND.SIGN is the *main procedure*, or *top level procedure*. It uses a number of other procedures. These procedures are called *subprocedures* of HOUSE.AND.SIGN. A good way to visualize the relationships among procedures is by using a *procedure tree*. In a procedure tree, a subprocedure is drawn below the procedure that uses it. For example, the procedure tree for HOUSE.AND.SIGN looks like this:



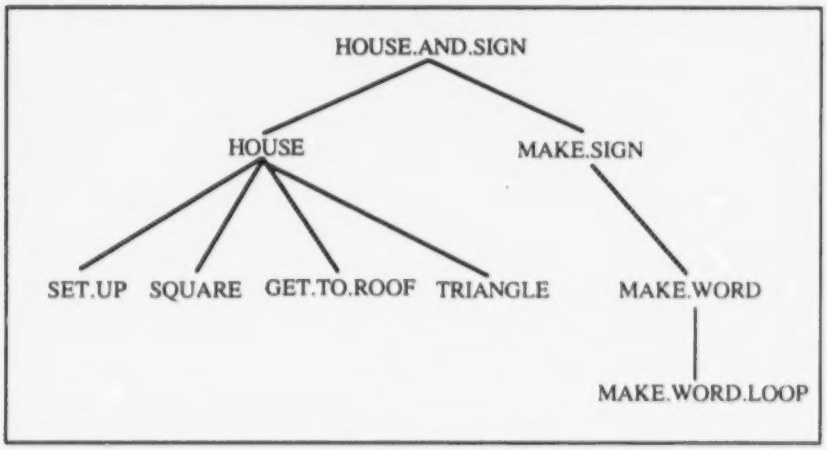
Notice that the HOUSE procedure, which is a subprocedure of HOUSE.AND.SIGN, has four subprocedures of its own. The word *superprocedure* is also commonly used to describe relationships among procedures. For example, HOUSE.AND.SIGN is a superprocedure of MAKE.SIGN.

To summarize:

- HOUSE.AND.SIGN is the main, or top level procedure.
- HOUSE and MAKE.SIGN are subprocedures of HOUSE.AND.SIGN.
- HOUSE.AND.SIGN is a superprocedure of HOUSE and MAKE.SIGN.
- SET.UP, SQUARE, GET.TO.ROOF, and TRIANGLE are subprocedures of HOUSE.
- HOUSE is a superprocedure of SET.UP, SQUARE, GET.TO.ROOF, and TRIANGLE.

Notice how much easier it is to see the relationships among procedures by using a procedure tree. The list of the relationships is hard to follow. You should *always* draw a procedure tree when you are working with more than two or three procedures.

If you use the MAKE.WORD procedures to stamp text, then you should also add these procedures to your procedure tree. Your procedure tree would then look like this:



Tips and Techniques

When you are working in the Editor, occasionally lines of your procedures go beyond the edge of the Editor Window. You can enlarge the window if you wish. However, you cannot press the Return key in the middle of a line. Return signals the end of a Logo instruction line. Instead, you can use the Tab key to continue a line. The Tab key adds an arrow to tell the person reading the procedure that the line continues below:

```
TO SQUARE
REPEAT 4 →          The Tab key was pressed to make this arrow.
      [FORWARD 50 RIGHT 90]
END
```

Mistakes in computer programs are called "bugs." The process of removing these errors is called "debugging." You will quickly learn that you spend more time debugging your Logo programs than you do writing them. Don't let that discourage you. Even professional computer programmers spend most of their time debugging. You should consider the time you spend debugging as part of the learning process.

The ideas in this section of each chapter have been designed to help you avoid or fix bugs when you were working in the immediate mode. Increasingly, this section of each chapter will include techniques for debugging. Read them carefully. They will save you a lot of time in your future work.

When you are working with a number of procedures, you should be sure each procedure works by itself. That is, after writing any one procedure, leave the Editor and type the name of the procedure. As you build up your program, test each procedure as you go. Thus, in the above example, after you tested SET.UP, SQUARE, TRIANGLE, and GET.TO.ROOF separately, you should test HOUSE to be sure all the pieces work together properly. Don't try to write your entire program all at once and then test it by typing the name of the top level procedure. There are simply too many things that can go wrong in a program that is more than two or three procedures long.

Give your procedures meaningful names, and keep your procedures quite short. It is easy to find yourself adding more and more commands to a procedure. Soon you have a very long

procedure that is hard to debug. Perhaps the name of the procedure no longer describes the function of the procedure. Take time to subdivide and rename procedures as you work. It will save a lot of time later when you are debugging.

Project Suggestion

Now that you know how to write multiple procedures, plan a picture of your choice and write the procedures to create it. Move along step by step like the example in this chapter. Draw a procedure tree as you go. Be sure to keep the individual procedures short. You should not have any procedures longer than 20 lines with one instruction on each line. If you put too much into a procedure, it becomes hard to find errors.

11. A Word About Designing Programs

There are a number of ways to go about creating programs. Different people prefer different approaches. Different computer languages lend themselves to different methods. Now that you are beginning to write programs containing a number of procedures, you should spend some time thinking about the best way to design a particular program.

Top Down Programming

Top down design refers to writing a program "from the top." Think about the HOUSE.AND.SIGN program from the previous chapter. If you use a top down approach, the first procedure you write will be

```
TO HOUSE.AND.SIGN
HOUSE
MAKE.SIGN
END
```

Next you write the HOUSE procedure:

```
TO HOUSE
SET.UP
SQUARE
GET.TO.ROOF
TRIANGLE
END
```

Then you write SET.UP, SQUARE, GET.TO.ROOF, and TRIANGLE. Finally, you write MAKE.SIGN. With this method, you plan the whole and then write ever smaller parts. This requires careful advanced planning and allows little room for exploration as you go.

Bottom Up Programming

In bottom up programming, you start with small pieces and build them up into a complete program. If you use bottom up programming in the HOUSE.AND.SIGN program, you don't start with a plan. You might first create the SQUARE procedure and then the TRIANGLE procedure. After some fiddling with these, you might discover that, when put together, they make a nice house. That might lead to the HOUSE procedure.

You might be experimenting with STAMPTEXT and get the idea to write a procedure to put a sign next to the house. Next would come the MAKE.SIGN procedure. Last of all, the HOUSE.AND.SIGN procedure would be written.

With this method, there is little advance planning. You play with an idea, turn it into a procedure, and let that idea lead you to another. You have little or no idea where you are going as you begin. Logo lends itself to the bottom up style of programming better than most programming languages. Languages like Pascal all but force programming in a top down style.

Much of what is written about Logo implies that the only "right" way to program in Logo is bottom up. This emphasis on bottom up style grows out of the philosophy of discovery learning that is often too closely allied with Logo. It is assumed that the Logo programmer will experiment with an idea and then use that idea as part of a larger project.

Middle Out Programming

Very few people actually program *either* in a true top down or bottom up style. Most people use some other method for writing programs, which can be described as "middle out." Middle out is the method that was actually used for writing the HOUSE.AND.SIGN program. There was an initial idea of what the program should do. Then small parts were written. Finally the whole thing was put together. Some things were done top down, other things bottom up, but most things were done from the middle, working both toward the top and the bottom.

Polishing Your Program

It doesn't really matter what programming style is used to do a project. However, it is best if the final program looks as if it were done top down. That is, there should be a top level procedure made up primarily of procedure names whose titles describe the parts of the program. Many of the other procedures in the program may also be made up of procedure names that describe the smaller parts.

Both the programming process *and* the final product are important in any programming project. During the development of a project, you probably focus on the debugging process. However, when you have finished a program in Logo, it should be polished for publication, just as you polish your final draft of a paper before you turn it in for your English class.

What is the correct style to use when writing a Logo procedure? Are there "rules" you should follow? The issue of programming style is one for which there is very little agreement within the Logo community. At one end of the spectrum are those who feel that to impose *any* rules for writing procedures is inappropriate. They feel that such rules will interfere with creativity and exploration. This book uses a very consistent and somewhat rigid style. This approach lies at the other end of the spectrum.

Any Logo programs you write should be polished for "publication," no matter whom you give them to. Here are some guidelines to help you begin developing a readable programming style.

1. Procedures should have meaningful names, that is, the name of the procedure should describe what happens in that procedure.
2. There should be no more than one instruction per line.
3. Each procedure you write should fit in the Edit Window. Even though you can size this window, it is still a good idea to limit the length of your procedures to fewer than 15 or 20 lines.
4. The main or top level procedure should contain only user-defined procedure names. That is, you should not use a Logo primitive in the main procedure. In the HOUSE.AND.SIGN example above, you saw that a SET.UP procedure was used to prepare the screen for use:

```
TO SET.UP
CLEARGRAPHICS
PENUP
BACK 40
PENDOWN
END
```

5. The only procedures in memory should be procedures used by the program.
6. When you use procedure inputs or global names, you should use meaningful words that describe the function of the input or name. (Inputs and naming will be discussed later in this book.)

These rules are perhaps unnecessarily rigid for your long-term work with Logo programming. However, they are designed to give you specific guidelines to follow from the outset that will help you in debugging your future, more complex programs. As you continue your work with Logo beyond this book, you will no doubt relax or modify these rules in ways that are comfortable for you.

Programs written using the above guidelines are much easier to work with. Other people can easily tell what each part does. You can more easily make changes in your own work at a later date. Most important, though, is the way in which a program written in top down style is easy to "debug." (Things in programs that don't work the way you want are called "bugs.") Sometimes the bugs are easy to find, sometimes they aren't. If the program is written well, then the debugging process takes less time.

Project Suggestions

Select a project that involves writing a number of procedures. As you are working on it, examine your own programming style. Which of the styles mentioned above best describes the way you prefer writing programs? Be sure your final program looks as if it were done in a top down manner.

12. Using PRINT

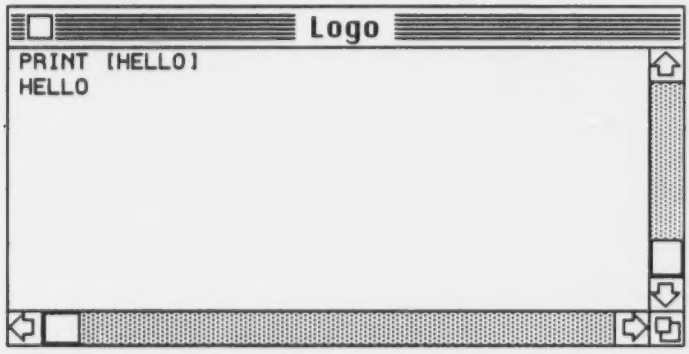
You have seen two forms of text when working with *Terrapin Logo*. There is the text you type in the Logo Window and there is the text you put in the Turtle Window using STAMPTEXT. The text in the Turtle Window, is really part of the graphics drawn by the turtle. It can't be manipulated once it is stamped on the screen.

You can also put text in the Logo Window by using commands. First be sure your cursor is in the Logo Window. Then you can type

```
PRINT [HELLO]
```

or

```
PR [HELLO]
```



The word "HELLO" appears below the PRINT command in the Logo Window. If you type

```
PRINT [Hello]
```

then the screen will look the same as above. It doesn't matter whether you use upper- or lowercase after PRINT. Only uppercase letters are printed.

You can also use PRINT from a procedure. Go to the Editor and type

```
TO GREET
CLEARTEXT
PRINT [GREETINGS, MARY!]
END
```

Now, leave the Editor with Enter to define the procedure and type

```
GREET
```

Then,

```
GREETINGS, MARY!
```

appears at the top of the Logo Window.

The CLEARTEXT command removes all the text from the Logo Window. Each time you issue a PRINT command, Logo prints what follows and then moves the cursor to the next line. Thus, you can write a procedure to create a more elaborate greeting as follows:

```
TO GREET
CLEARTEXT
PRINT [GREETINGS, MARY!]
PRINT [HOW ARE YOU?]
PRINT [I'M FINE!]
END
```

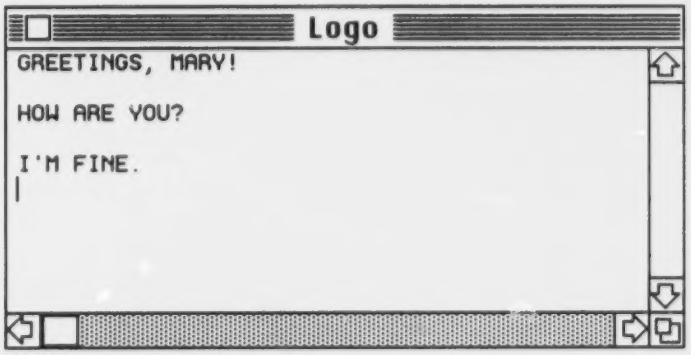
When you run this procedure,

```
GREETINGS, MARY!
HOW ARE YOU?
I'M FINE!
```

appears on the screen.

What if you want some blank lines between the parts of your greeting? You can add blank lines by printing an empty line:

```
TO GREET
CLEARTEXT
PRINT [GREETINGS, MARY!]
PRINT []
PRINT [HOW ARE YOU?]
PRINT []
PRINT [I'M FINE!]
END
```



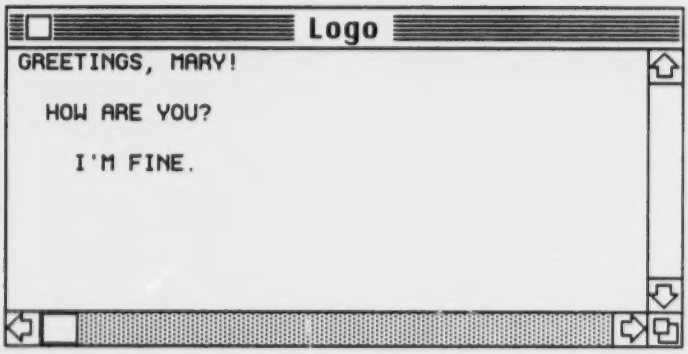
Now your message is double spaced.

What if you want to put some spaces in front of some of the lines? You can do this by inserting blanks. There are several ways to do this. The clearest method is to use CHAR 32. CHAR 32 is simply a way to represent a space. Thus, you might enter

```
TO GREET
CLEARTEXT
PRINT [GREETINGS, MARY!]
PRINT []
(PRINT CHAR 32 [HOW ARE YOU?])
```

```
PRINT []
(PRINT CHAR 32 CHAR 32 [I'M FINE!])
END
```

If you type GREET, you now see



Did you notice the parentheses ( ) around the last two PRINT instructions? These allow PRINT to have more than one input. In this case both CHAR 32 *and* the list to print are inputs to PRINT. You can have as many inputs to PRINT as you want by using parentheses in this way.

Inserting lines with many PRINT [] instructions and inserting spaces by adding a lot of CHAR 32s in the PRINT line is a bit cumbersome. Instead, you might want to create some other "tool" procedures. The following procedure inserts any number of lines:

```
TO LINES :NUMBER
REPEAT :NUMBER [PRINT []]
END
```

The procedure given below inserts any number of spaces:

```
TO SPACES :NUMBER
REPEAT :NUMBER [PRINT1 CHAR 32]
END
```

Notice that this procedure used the PRINT1 command. PRINT1 works exactly like PRINT except that it does not move the cursor to the next line. Thus, if you enter this procedure,

```
TO TEST.PR1
PRINT1 [PART 1]
PRINT1 [PART 2]
PRINT [PART 3]
END
```

and type

```
TEST.PR1
```

you see

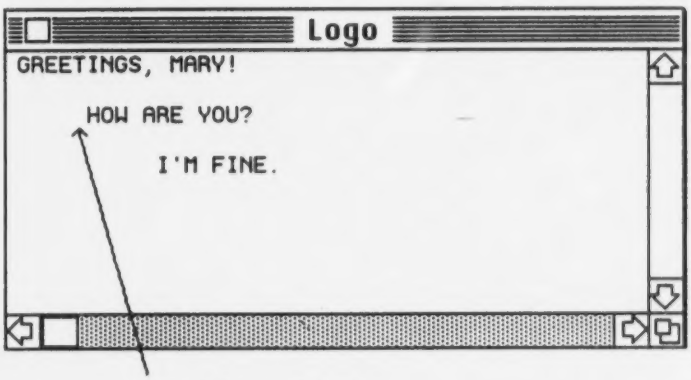
```
PART 1 PART 2 PART 3
```

The cursor has moved to the next line because the last instruction in the TEST.PR1 procedure is a PRINT command. You will no doubt find PRINT1 useful as you learn more about working in the Logo Window.

Now you can use LINES and SPACES to place text in the Logo Window:

```
TO GREET
CLEARTEXT
PRINT [Greetings, Mary!]
LINES 1
SPACES 5
PRINT [How are you?]
LINES 1
SPACES 10
PRINT [I'm fine!]
END
```

You see



5th column, 3rd line

Tips and Techniques

If you know another programming language, such as BASIC or Pascal, take special note of the use of parentheses with PRINT. The left parenthesis goes before the word PRINT, not after it. You will find that a lot of articles and books about Logo use SENTENCE (see chapter 40) instead of parentheses in order to print more than one thing. This is because some early versions of Logo did not allow the use of parentheses with PRINT.

There are a couple of ways you can print lowercase letters in *Terrapin Logo*. If you used the *string* to put more than one word in the Turtle Window using STAMPTEXT, then you will find that you can use strings in the same way with PRINT. Thus, typing

```
PRINT `hello`
```

causes

```
hello
```

to be displayed. If you used MAKE.WORD to stamp text, then all you need in addition is to know that *Terrapin Logo* has a *reporter* that allows you to print in lowercase letters. You can type

```
PRINT LOWERCASE "HELLO
```

and you see

```
hello
```

One quotation mark to the left of the word is needed here. (You will learn more about the meaning of this symbol later in this book.) However, you can use the MAKE.WORD procedures you used with STAMPTEXT to make a number of words into lowercase. Thus, if you type

```
PRINT LOWERCASE MAKE.WORD [THIS IS A TEST]
```

you see

```
 this is a test
```

The extra initial space is inserted by MAKE.WORD. You can eliminate that first space by typing

```
PRINT BUTFIRST LOWERCASE MAKE.WORD [THIS IS A TEST]
```

You will learn about BUTFIRST (and other powerful reporters for manipulating words and lists) later in this book.

Project Suggestions

Write a procedure to put a poem spaced neatly on the screen. Make use of blank lines and leading spaces.

Write a procedure to paint a "word picture." It might be a pun, like

```
 MAN  (man overboard)
BOARD
```

or something more descriptive like

```
FALLING D
         O
           W
             N
```

13. Managing Your Workspace and Files

All the procedures you define are in the computer's memory. This collection of procedures is part of the *workspace*. Later, you will learn what other Logo objects are in the workspace. As long as you don't turn off the computer, all your procedures remain available to you. On the other hand, procedures you no longer want also become part of your workspace. There are a number of commands you need to learn in order to manage your workspace.

Commands to Manage Your Workspace

Suppose you have defined three procedures: SQUARE, TRIANGLE, and CIRCLE. You can edit one of them by typing

```
EDIT "CIRCLE
```

or you can edit all of them by typing

```
EDIT [SQUARE TRIANGLE CIRCLE]
```

However, if you only want to know the names of all the procedures in your workspace, you can type

```
POTS
```

This stands for Print Out TITLES. With the three procedures mentioned above, typing

```
POTS
```

gives

```
TO SQUARE
TO TRIANGLE
TO CIRCLE
```

You see just the title lines of all the procedures in your workspace.

If you want to examine a particular procedure (but not edit it), you can type

```
PO "SQUARE
```

and Logo prints all the commands in procedure SQUARE. If you want to see all the procedures in your workspace, you can type

```
PO PROCEDURES
```

for Print Out PROCEDURES. If you want to see everything in your workspace, you can type

```
POALL
```

for Print Out ALL. Logo prints all the procedures in your workspace as well as other things that you will learn about later, such as global names.

Sometimes you will want to get rid of procedures you no longer want. To do this, you can use ERASE (short form ER). For example,

```
ERASE "SQUARE
```

removes the procedure SQUARE from the workspace, and

```
ERASE [SQUARE TRIANGLE]
```

removes the procedures SQUARE and TRIANGLE from the workspace. If you want to remove everything from your workspace, then you can use

```
ERALL
```

for ERase ALL. Use these commands with care. Unless you have previously saved your procedures, once you use erase commands, they are gone from memory. There is no way to get them back.

Tips for Managing Your Workspace

When you are programming in Logo, it is very important that you always be aware of what is in your workspace. You should not let it get cluttered with unwanted or unnecessary procedures. Take the time to keep your workspace orderly at all times. Here are some important suggestions:

1. Before you begin working, especially if someone else has been using the computer, type

   ```
   ERALL
   ```

   to be sure the computer's memory is clear.

2. Every 10 to 15 minutes, type POTS to see what procedures you have defined, and use ERASE to remove any that you no longer want.

3. When you leave the computer, type

   ```
   ERALL
   ```

   so that the next person using the computer has no procedures in memory when s/he begins. Otherwise, someone else can end up with your work.

Working with Files

You have learned how to group Logo instructions into procedures so that they can be used again. In addition, you have seen that all the procedures you write become part of your workspace. However, once you turn off the computer, all your work is gone. Recall that you used SAVEPICT to save your picture. There is a similar command, SAVE, that saves everything in your workspace.

You used a disk to save your pictures. You can use the same disk to save your workspace. Suppose your workspace contains three procedures. For example, if you type

```
POTS
```

then you see

```
TO SQUARE
TO TRIANGLE
TO CIRCLE
```

Now, with your own disk in the disk drive, type

```
SAVE "TRIAL.1
```

Everything in your workspace is saved on the disk under the name TRIAL.1. In order to see what files are on the disk, pull down the File menu and choose either Load or Save. You can then scroll through the list of files on the disk. In this case you would see the name TRIAL.1, among other things.

At this point, your three procedures are on the disk *and* in your workspace. If you then type

```
ERALL
```

the procedures are only on the disk. They are erased from the workspace but not from the disk.

When you want to use these procedures again, you can load them into your workspace by typing

```
LOAD "TRIAL.1
```

or by pulling down the File Window, selecting Load, and choosing TRIAL.1. The procedures are then in the memory of the computer again.

What if you no longer want a file on your disk? You can erase it by using the ERASEFILE command. For example, if you want to erase TRIAL.1, then you type

```
ERASEFILE "TRIAL.1
```

Be very careful when using this command. Once a file is removed from the disk, it is gone forever.

When you begin saving files, it is a good idea to establish a plan for saving. When you use the same name, Logo asks you if you want to replace the file currently on disk. If you say "yes," then whatever is currently in the memory of the computer replaces what is in the file on disk. If you are not very careful, you can easily erase procedures you want to keep.

Establish a plan for saving. A good method to use is as follows:

1. Begin by typing ERALL.
2. Work for 10-15 minutes.
3. Clean up your workspace, that is, get rid of any unwanted procedures.
4. Save your work using a name like TRIAL.1
5. Continue to repeat steps 2 through 4, but use successively higher numbers for your files: TRIAL.1, TRIAL.2, TRIAL.3, etc.

6. When you are finished, choose a final name for your work, save it on your files disk, and if possible, on a second disk, called a backup disk.

If you have to stop for any reason, the next time you want to work you can simply load that file with the highest number. If, as you are working, you decide you don't like your most recent revisions, you can return to the next lower number and begin again.

Workspace and Files

There are several important facts you should know about the relationship between the workspace and files, although they may not all seem meaningful just now. You should return to this list from time to time as your understanding of Logo grows.

1. When you use SAVE followed by a file name, then everything in your workspace is saved in that file on disk. You need only save *once* to save all the procedures you have defined.

2. When you type ERALL, everything is removed from the workspace but not from the disk. You will learn how to erase files from the disk in the next section.

3. If you already have procedures in the workspace and load another file, the procedures in that file will be added to the ones in the workspace, that is, you will have all the procedures that were in memory before you read the file and, in addition, you will have all the procedures in the file.

4. If you have a procedure in the workspace with the same name as a procedure in a file on the disk, and then you load that file, the procedure on the disk will replace the one in the workspace, that is, the procedure in memory will be gone and the procedure from the file will completely replace it.

Project Suggestion

There is important material in this chapter. Spend time reviewing it, perhaps trying each of the new commands as you learn them. Even if you review this chapter now, return to it later when you have worked with Logo for a longer time.

14. Using Pen Modes

Earlier in this book you learned how to change the way the turtle draws by changing the pen it carries. You used SETPENSIZE to change the width and the height of the pen. You also learned that you could tell the turtle to draw in any of 38 different pen patterns by using SETPPATTERN. In addition, you have learned that the turtle can move with its pen up or with its pen down. At the very beginning of your work with Logo, you learned to use PENUP to lift the pen and PENDOWN to lower the pen.

You also saw that PENERASE could be used to remove lines you had drawn and that PENPAINT was used to put the turtle "back to normal." Both PENERASE and PENPAINT are *pen modes*. PENERASE is clearly useful if you want to erase a line. Thus, if you type

```
REPEAT 4 [FORWARD 50 RIGHT 90]
```

and then decide you no longer want the square, you can type

```
PENERASE
REPEAT 4 [FORWARD 50 RIGHT 90]
```

to erase the square. You can save yourself some typing, however, by first typing PENERASE (PE), then using the arrow keys or the mouse to move to the REPEAT line, and then pressing Enter. To get the turtle back to its normal drawing mode, you then type

```
PENPAINT
```

PENREVERSE

Yet another pen mode is PENREVERSE (PX). Begin by typing

```
PENPAINT
SETPPATTERN 1
FILLSH [REPEAT 2 [FORWARD 20 RIGHT 90 FORWARD 100 RIGHT 90]
```

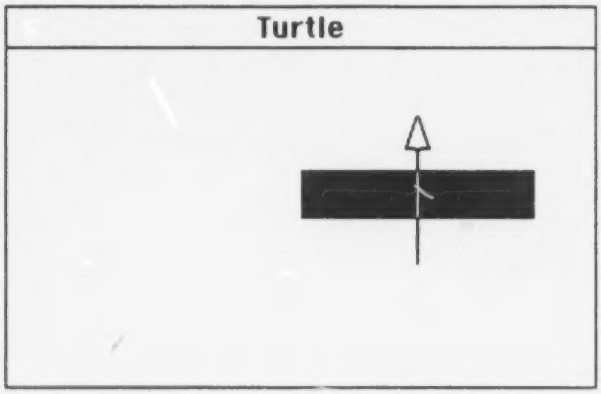
These commands make sure you begin with the pen in its normal mode. Black is selected as the pen pattern. Finally, a rectangle is drawn and filled with black. Next type

```
PENUP
BACK 20
RIGHT 90
FORWARD 50
LEFT 90
PENDOWN
```

This places the turtle below the black rectangle. Finally type

```
PENREVERSE
FORWARD 50
```

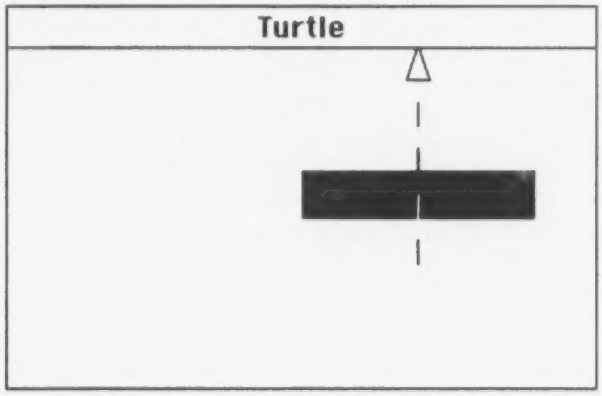
You see



Notice that the pen turns white where it goes over black dots but stays the same where it goes over white dots.

Next, repeat the series of instructions given above to draw a black rectangle and place the turtle. (You may want to write a SET.UP procedure.) Then type

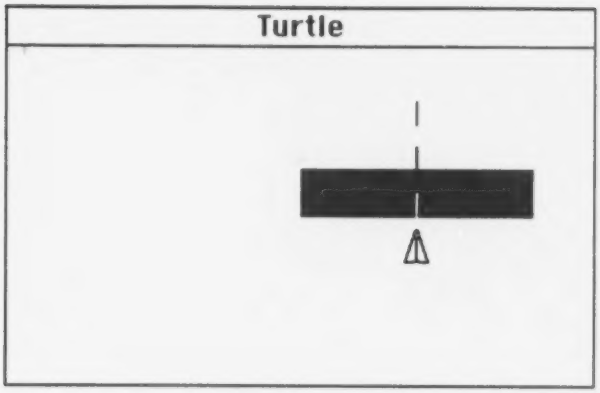
```
PENREVERSE
REPEAT 4 [PENDOWN FORWARD 10 PENUP FORWARD 10]
```



Do you see that black dots become white, and white dots become black? Next type

```
BACK 80
```

Then you see



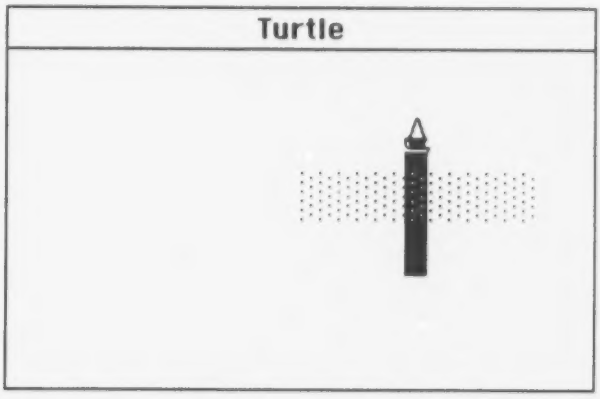
Next, draw a rectangle just as you did above, but this time use pen pattern 21 for the pen pattern. After drawing the rectangle, type

```
SETPPATTERN 1
SETPENSIZE 10 10
```

to get a black pen that is 10 pixels on a side. Finally, type

```
PENREVERSE
FORWARD 50
```

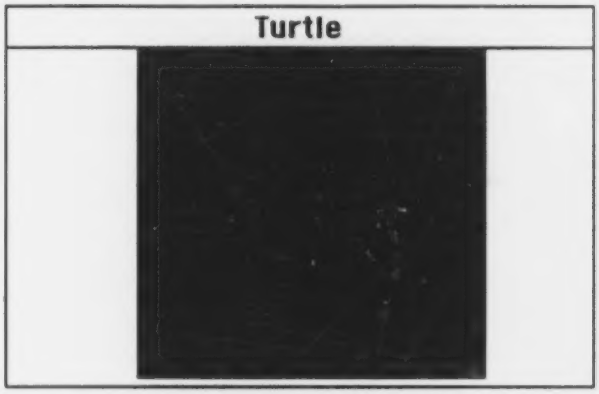
You again see that the black pixels are turned to white and the white pixels are turned to black.



You can achieve some interesting effects with PENREVERSE. If you fill a large square with black using pen pattern 1, place the turtle, and then type

```
SETPENSIZE 5 5
SETPPATTERN 29
PENDOWN
PENREVERSE
REPEAT 5 [FORWARD 100 RIGHT 144]
```

you see



PENMIX

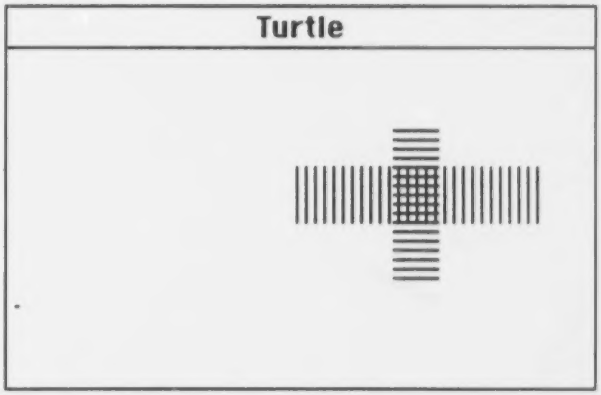
Yet another pen mode is PENMIX. This mode causes the pen pattern to mix with the background pattern. Again, draw a rectangle, but this time use pen pattern 8:

```
PENPAINT
SETPPATTERN 8
FILLSH [REPEAT 2 [FORWARD 20 RIGHT 90 FORWARD 100 RIGHT 90]
```

Move the turtle to below the rectangle as before. Then type

```
SETPENSIZE 20 20
SETPPATTERN 27
PENMIX
FORWARD 50
HIDETURTLE
```

You see



Notice that the pattern of the pen and the pattern of the background are now mixed together. Like PENREVERSE, PENMIX can be used to achieve some interesting effects. For example, begin by drawing a rectangle using pen pattern 31:

```
PENPAINT
SETPPATTERN 31
SETPENSIZE 70 1
FORWARD 70
```

Then move the turtle so that it is above the bottom of the rectangle in the Turtle Window:

```
PENUP
BACK 60
```

Change the pen pattern to 13:

```
SETPPATTERN 13
```

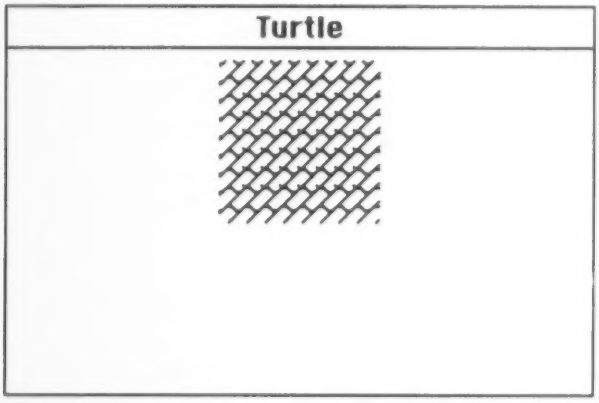
Make the pen size smaller and use PENMIX:

```
SETPENSIZE 50 1
PENMIX
```

and finally draw

```
PENDOWN
FORWARD 50
HIDETURTLE
```

You see



The combinations of patterns you can achieve are limited only by your imagination.

Tips and Techniques

You can't tell by looking at the turtle what pen mode you are using. It is easy to get confused when switching among pen modes. *Terrapin Logo* provides the reporter PENMODE that associates a number with each mode. If you type

```
PENPAINT
PRINT PENMODE
```

you see the number 0. Each of the modes you have learned has a number associated with it:

PENPAINT	0
PENMIX	1
PENREVERSE	2
PENERASE	3

You can also use the command SETPENMODE to set a pen mode. For example, typing

```
SETPENMODE 2
```

is the same as typing PENREVERSE.

There are actually eight possible pen modes you can use. The short table below summarizes the effect of each.

Pen Mode Setting	If the pen pattern pixel is black:	If the pen pattern pixel is white:
SETPENMODE 0 (PENPAINT)	Force black	Force white
SETPENMODE 1 (PENMIX)	Force black	Leave alone
SETPENMODE 2 (PENREVERSE)	Invert*	Leave alone
SETPENMODE 3 (PENERASE)	Force white	Leave alone
SETPENMODE 4	Force white	Force black
SETPENMODE 5	Leave alone	Force black
SETPENMODE 6	Leave alone	Invert*
SETPENMODE 7	Leave alone	Force white

*Invert means:

- if a pixel is black in the pen pattern, then reverse the pixel in the background
- if a pixel is white in the pen pattern, then leave the pixel alone in the background. ("Reverse" means make a black pixel white or make a white pixel black; background means the pixels are under the pen being carried by the turtle.)

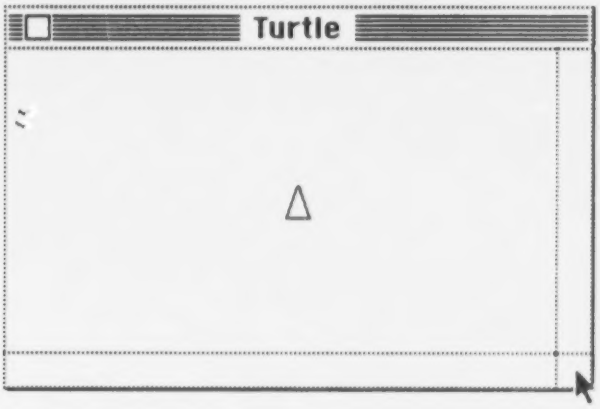
Project Suggestions

Experiment with different pen modes and combinations of pen patterns, modes, and sizes. What kind of interesting effects can you achieve?

15. Using Windows

As you have been using *Terrapin Logo*, you no doubt noticed that the standard windows appear in the same places every time you start Logo. You can move any of the windows to different spots on the screen by clicking on the Title Bar and dragging. You can size both the Edit and Logo windows in the usual manner. In addition, *Terrapin Logo* allows you to size the Turtle Window and to manipulate any kind of window by using commands.

Click on the bottom right corner of the Turtle Window. You see



Now you can make the window longer or narrower by dragging down or to the left. If you want the window to be wider, first click on the Title Bar and then drag the window to the left. Then click in the lower right corner and size the window by dragging to the right.

To create a new graphics window, begin by typing

```
MAKEWINDOW "GRAPHICS "TEMP
```

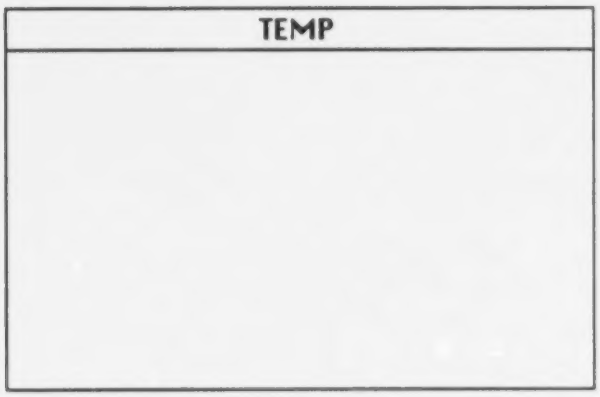
This creates a new graphics window. In order to see this window, either use your mouse to move the Turtle Window or type

```
CLOSEWINDOW "TURTLE
```

The standard Turtle Window disappears. Next type

```
SHOWTURTLE
```

You see



Notice the name at the top of the window.

You can create three types of windows: edit windows, graphics windows, or text windows. As you create a new window, it will appear "behind" the current window. Thus, if you type

```
MAKEWINDOW "TEXT "NEWTEXT
```

nothing appears to happen. However, if you close the Logo Window, you see your "NEWTEXT" window. You can now enter commands in the new text window.

In addition to creating and closing windows, you can bring a window to the front with a command. For example, if you type

```
MAKEWINDOW "EDIT "MYEDIT
```

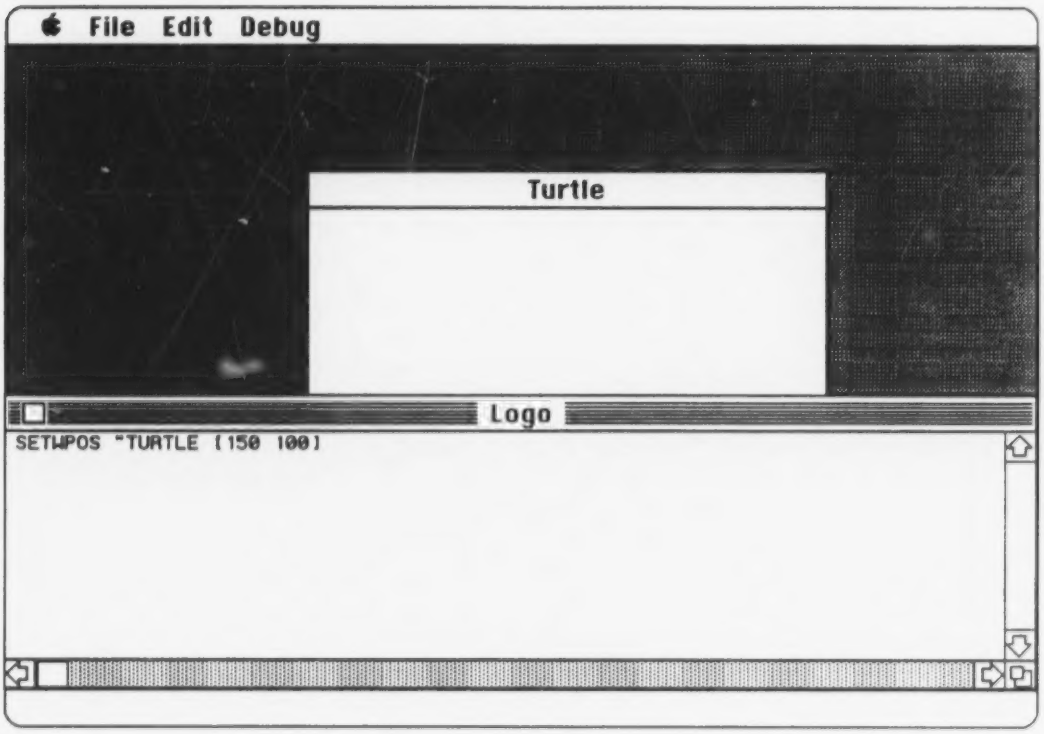
to create a new edit window. You can bring it to the front by typing

```
SETACTIVEW "MYEDIT
```

Perhaps you have been frustrated by the size or placement of the Turtle Window. You can use Logo commands to change this (and other) windows. Begin by typing

```
SETWPOS "TURTLE [150 100]
```

You see

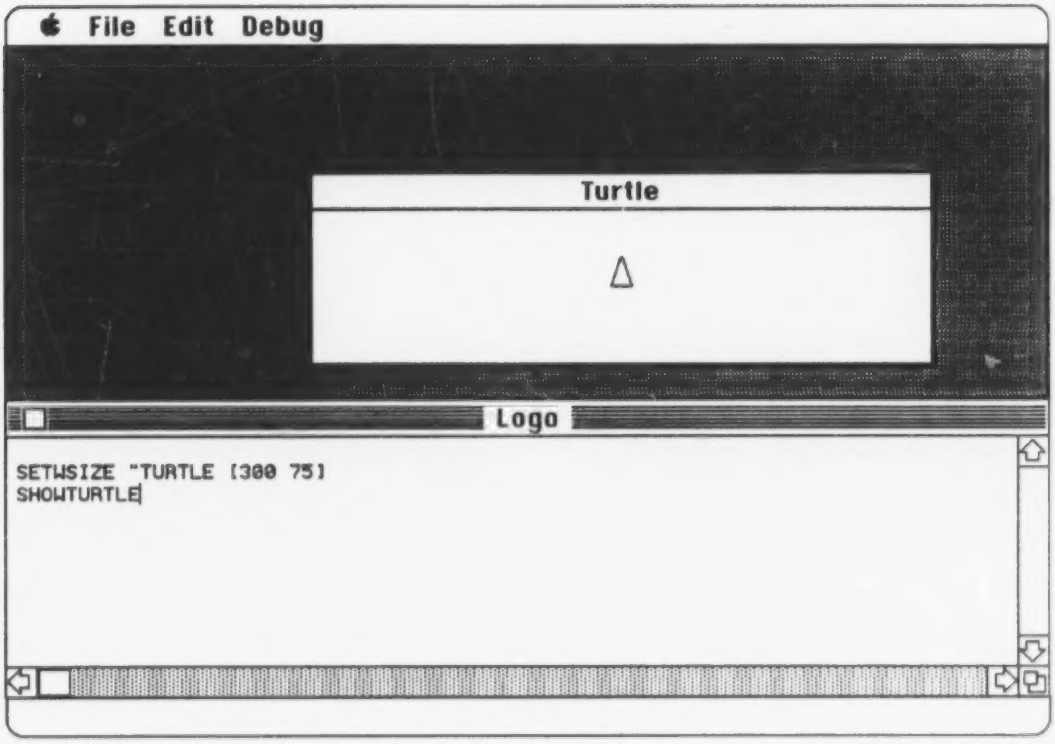


The window has moved from its normal position on the screen. The numbers given after SETWPOS represent the coordinates of the upper left corner of the window. The screen is numbered from left to right and top to bottom.

Next, after typing

```
SETWSIZE "TURTLE [300 75]
SHOWTURTLE
```

you see

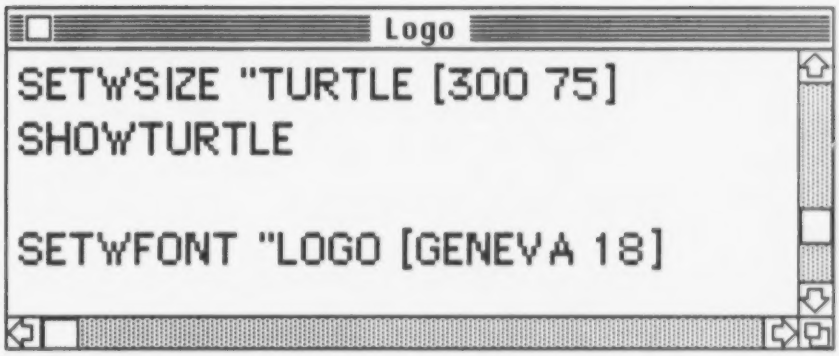


The numbers after SETWSIZE are the width and the height of the window. With SETWPOS and SETWSIZE you can place the Turtle Window anywhere on the screen. This can be quite useful when you write a procedure in which you want a larger Turtle Window than Logo normally provides.

Finally, you can change the font used in any text window. SETWFONT requires the name of the window and then both the font name and size. Thus, if you type

```
SETWFONT "LOGO [GENEVA 18]
```

you see



The window is redrawn in the new font. Remember that the font you use must be in the system you use when you start Logo.

Tips and Techniques

As you work with windows, it is easy to become confused or lost. If you simply can't figure out what happened, Quit Logo and restart. That will give you fresh Logo, Turtle, and Edit windows. You can then experiment with the knowledge that you have exactly three windows available.

There are more window commands and reporters available than are mentioned in this chapter. If you are interested in more sophisticated work with windows in Logo, you should read the *Terrapin Logo* Reference Manual.

Project Suggestions

Spend some time experimenting with windows. You should particularly focus on being able to change the size and position of the Turtle (or graphics) windows. What happens to a Logo drawing that wraps when you make the window larger? Is the "center" of the window always the same regardless of the position or size? How big can a window be? How small?

16. Multiple Turtles

By now you have done a lot of drawing with the turtle. You know that you can create designs; change pen sizes, patterns, and modes; and stamp lots of different shapes using the turtle. But *Terrapin Logo* has more than one turtle. So far you have seen only turtle number 0.

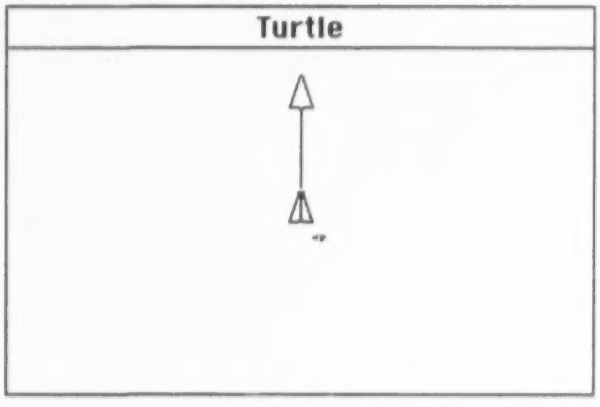
When you start Logo, you always see the familiar turtle, turtle 0. Now type

```
TELL 1
SHOWTURTLE
```

The turtle seems to disappear. Type

```
FORWARD 50
```

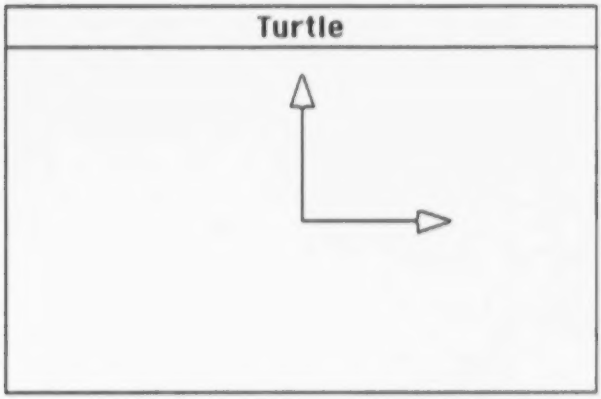
You see



The turtle near the top of the window is turtle number 1. The turtle at the center is turtle number 0. Try typing

```
TELL 0
RIGHT 90
FORWARD 50
```

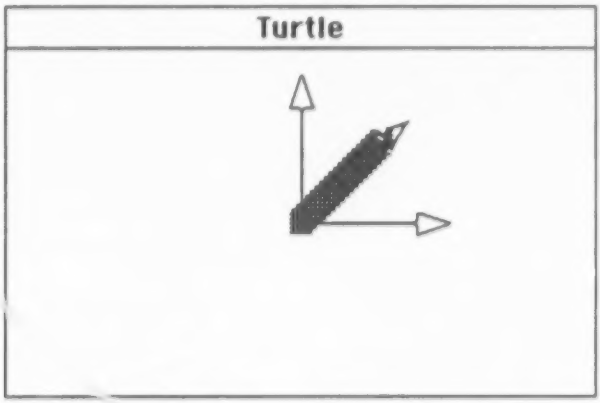
You see



Notice how each turtle moves only when "spoken to" with the TELL command. Next type

```
TELL 2
SHOWTURTLE
SETPENSIZE 10 10
SETPPATTERN 5
RIGHT 45
FORWARD 50
```

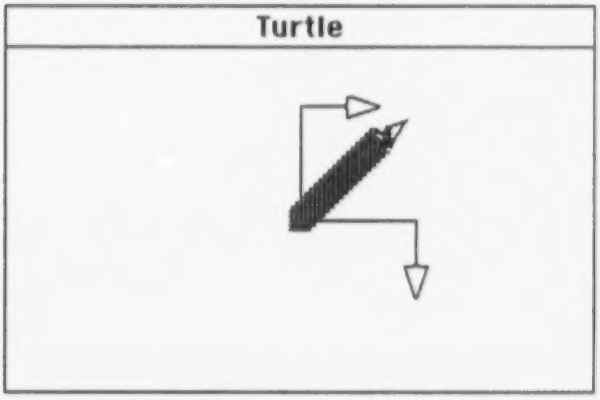
You see



You can have 32 turtles in *Terrapin Logo*. You can talk to each turtle using the TELL command. You can also talk to more than one turtle at a time. Try typing

```
TELL [0 1]
RIGHT 90
FORWARD 20
```

Now you see



Notice that only turtle number 0 and turtle number 1 moved. You can talk to any combination of turtles by putting their numbers inside square brackets. Be careful to use [and], not parentheses (and).

Try writing a simple procedure, such as

```
TO SQUARE
REPEAT 4 [FORWARD 20 RIGHT 90]
END
```

Then type

```
TELL [0 1 2]
HOME
CLEARGRAPHICS
SETPENPATTERN 1
SETPENSIZE 1 1
SHOWTURTLE
```

The HOME command returns all the turtles to the center of the screen, facing up. The other commands clear the screen and give all the turtles the "normal" pen size and pattern. Next give different instructions to some of the turtles:

```
TELL 1
RIGHT 45
TELL 2
RIGHT 90
```

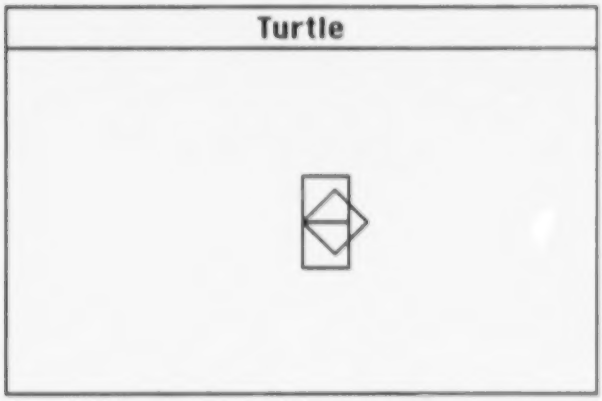
Your three turtles now form a fan-shape at the center of your Turtle Window. Finally, activate all your turtles by typing

```
TELL [0 1 2]
```

Then type

```
SQUARE
HIDETURTLE
```

You see



It can be a lot of fun to have a number of turtles active at once! However, you should keep in mind as you are working with multiple turtles that it doesn't make sense to use several turtles just because they are available. For example, there is no reason to use two different turtles to stamp different shapes in the Turtle Window. On the other hand, if you want two turtles *active* at the same time, then you must use more than one turtle. You will learn more reasons for having multiple turtles when you learn about animation (Chapter 20).

Tips and Techniques

The most important concept to keep in mind when you begin working with multiple turtles is the meaning of the TELL command. TELL "wakes up" a turtle. It does nothing else. TELL does not make the turtle appear. You must type SHOWTURTLE. It does not make the turtle draw. You must use FORWARD or BACK. It does not turn the turtle. You must use RIGHT or LEFT.

You also need to keep in mind that a TELL command remains in effect until you use another TELL command. If you use TELL [1 3] then turtles 1 and 3 will obey *all* turtle commands that you use until you "wake up" other turtles with a different TELL command.

Remember that you must use TELL 0 to get the "regular" turtle active again.

It is easy to become confused when working with multiple turtles. You can easily lose track of which turtle you are using. WHO gives you the numbers of the active turtles. If you type

```
PRINT WHO
```

then a list of active turtles appears in the Logo Window. For example, if you type

```
TELL [1 3]
```

and then type

```
PRINT WHO
```

you see

```
[1 3]
```

You can put PRINT WHO in a procedure using multiple turtles to help in your debugging.

Project Suggestion

There are many interesting things you can do with multiple turtles. You can create fascinating patterns using some or all of the turtles at once. Try writing a procedure to create an interesting graphics design. Then have all the turtles draw it at once. You may want to write a SET.UP procedure that places each of the turtles before they begin drawing.

17. Logo Grammar

Until now you have been writing Logo code at a rather intuitive level. You have been given no syntax rules as guidelines. Instead, you have followed the models given and have experimented as needed, trying to understand error messages that appeared when you made a mistake. This is the way most people, both children and adults, begin their learning of Logo. Initially, it is quite easy to proceed this way. Logo was designed to provide an easy early entry point.

However, to have a more complete understanding of the Logo programming language, you need to understand its basic rules of grammar and punctuation. Once you understand these rules, you will find that learning new Logo primitives is easy, and that you can pick up a technical reference manual for any version of Logo and understand how each primitive works. This level of understanding is necessary if you are going to progress beyond the beginning level in Logo programming. Take note that everything in this chapter applies to nearly any version of Logo you encounter—it is in no way specific to *Terrapin Logo*.

On the other hand, you can't expect to completely understand these rules on first reading–or even on second reading! You should return to this chapter of the book from time to time. Gradually you will see how each primitive and concept that you learn fits into this larger picture.

Logo Objects

There are three kinds of "objects" that make up the Logo language: words, numbers, and lists. These three familiar terms have the following technical meanings in Logo:

word: an ordered collection of characters

number: a special kind of word that does not have to be quoted

list: an ordered collection of words and/or lists

The definition of *word* is not unlike its common meaning in English. However, a Logo word *can* contain symbols, such as &, >, and +. In practice, however, you will most often see Logo words containing just letters, numbers, and periods. Most of the time, a space is used to separate two different Logo words. The name of any Logo procedure must be a Logo word. For example, SQUARE or MOVE.TO.ROOF are both legal names of procedures. However, a procedure *cannot* have the name MY HOUSE because a space separates the title into two separate Logo words.

Note: You'll recall that we used *strings* to put mixed case letters on the screen by using either STAMPTEXT or PRINT. Strings are special objects in *Terrapin Logo* that generally do not appear in other versions of Logo. However, strings are really just special words. By using the back quote marks (`) around a group of words, you turn the group of English words into a single Logo word that includes the spaces.

The definition for *number* contains yet another technical term. To *quote* a Logo object means to cause it to be evaluated to itself. That is, nothing changes when Logo encounters a quoted object. For example, if you type

```
PRINT "HI
```

you are telling Logo to print on the screen the result of "HI. Since the quotation mark is the symbol for quoting a word, HI is evaluated to itself, and the word HI is printed on the screen. If, however, you type

```
PRINT HI
```

Logo responds with "There is no procedure named HI" because it *evaluates* HI and thus looks for a *procedure* named HI.

This idea of quoting is prevalent both in English and in other programming languages. When you say, "The word 'cat' occurs 10 times in this paragraph," you are referring to the symbols c-a-t, not the soft, furry animal. If you type

```
PRINT "Hello there!"
```

in BASIC or

```
WRITELN ('Hello there!');
```

in Pascal, you want the characters between the quotation marks printed exactly as they appear. In English and BASIC, we use quotation marks " ", in Pascal, we use ' ', and in Logo we use " before a Logo word and [ ] to surround a Logo list. More information about these punctuation marks will be given later.

Now, think again about the definition of a Logo number as being a special kind of word. In Logo you can type

```
PRINT 234
```

and the number 234 is printed, exactly as if you had typed

```
PRINT "234
```

The message "There is no procedure named 234" does not appear. You don't have to quote numbers in Logo. This special case exists so that Logo can use standard arithmetic notation. That is, you can type

```
PRINT 5 + 7
```

instead of

```
PRINT "5 + "7.
```

Finally, consider the third term in this section: *list*. You may have noticed something strange about this definition. It breaks the rule that you were no doubt taught about not using the word you are defining in the definition of that word. The definition of a Logo list is a special kind of definition. It is a *recursive definition*—a definition in terms of itself. You will learn more about this idea of recursion later. To give you an intuitive feel for what lists look like, here are some examples of *quoted* lists:

```
[cat dog rat hat]   <— a list of four words

[23 j qweklfsoifalko] <— a list of three words

[cat [dog] rat] <— a list of two words and one list (containing one word)

[[cat dog][rat mouse][house hat top]]<— a list of three lists
```

Logo Punctuation

In the above discussion, there were punctuation marks like " and [and] used. Let's formalize their meaning as well as examine a couple of others frequently used in Logo:

" <— used to quote a word

[ ] <— used to quote a list

( ) <— used as grouping symbols: e.g., (PRINT "HELLO "GOODBYE) or PRINT (93 + 75 + 86) / 3

: <— used to designate the value associated with a name. This symbol will be discussed in Chapter 26 of this book.

Note: The back quote mark can be thought of as the symbol to quote a string— to turn a group of English words into a single Logo word.

Too often when people are learning Logo, they ask questions such as "Should I use brackets or parentheses here?" Perhaps you have asked those questions as you were debugging your programs. Initially, that is OK, but as understanding grows, the above question should be rephrased as "Do I want a quoted list or do I want to group some part of an instruction here?" As you are asking yourself questions when you debug future programs, try rephrasing your questions in terms such as these.

Categories of Procedures

There are two different categories into which all procedures fall. You have already learned that you can describe a procedure as either a primitive (something that Logo "knows") or as a user-defined procedure. Both primitives and user-defined procedures can be subdivided into two groups: *commands* and *reporters* (another word for *reporter* is *operation*).

- A *command* is a procedure that causes an effect.
- A *reporter* (or *operation*) is a procedure that "returns" a value to Logo.

The procedures you have written to date are all commands, but you have used several reporters and will be learning more soon. Consider these examples:

If you type

```
FORWARD 50
```

the turtle moves forward 50 turtle steps. FORWARD is a command; its *effect* is to move the turtle the number of steps given as input to FORWARD. However, if you type

```
RANDOM 50
```

Logo responds with "Result: 37" (or some other number less than 50). RANDOM is a *reporter*. It reports the result of evaluating RANDOM 50 to Logo. You must tell Logo what to do with this result. For example, you might type

```
FORWARD RANDOM 50
```

Note that every Logo instruction must begin with a command, and all other procedures in that instruction, be they user-defined or primitive, must be reporters.

Finally, there is a simple way to understand the workings of any procedure, be it a primitive or user-defined. You need only answer these four questions:

1. Is it a command or a reporter?

2. How many inputs does it need?

3. What type or kind of Logo object (word, number, or list) must each of the inputs be?

4. If the procedure is a command, what is its effect?
   If the procedure is a reporter, what is its output–what does it report?

Practice these questions on some primitives or procedures you have written. (This series of questions comes from Chapter 2 of Brian Harvey's book *Computer Science Logo Style, Volume 1*, which you can read if you would like a more thorough discussion of Logo grammar.)

Summary

If you are a bit overwhelmed, don't worry. These ideas will become clearer as you do more work with Logo. In summary,

- everything in Logo is a *word*, *list*, or *number* (Strings are special words.)
- every procedure, be it user-defined or a primitive, is either a *command* or a *reporter*.

There are four kinds of punctuation marks used in Logo:

"	is used to quote a word
[ ]	are used to quote a list
( )	are used to group parts of an instruction
:	is used to refer to the value of a name

In addition, *Terrapin Logo* adds punctuation for a string:

`	is used to quote a string

Tips and Techniques

The material in this chapter is central to your understanding of Logo. It is unlikely that you fully understand the material in this chapter now. Understanding will come, however. Keep in mind that you should use the material in this chapter for reference, particularly as you encounter new primitives that you have difficulty understanding.

Project Suggestion

Select several Logo primitives you feel comfortable using. Can you answer each of the four questions found just before the Summary of this chapter? This is a good activity to do with a friend. You can check each other and discuss your disagreements. This kind of activity will help you understand Logo much better.

18. Using READCHARACTER

You have, no doubt, used a computer program that asked you a question and then responded to what you typed. Perhaps it asked for your name and then greeted you by name. Perhaps you chose a number from a menu. Programs that you "talk to" are called *interactive* programs. You can write interactive programs in Logo.

Recall that *Terrapin Logo* uses as many as 32 turtles. These turtles are numbered from 0 to 31. You can write a program that allows the user of the program to tell which turtle s/he wants to use. In order to do this, you need to use READCHARACTER. READCHARACTER is a *reporter*. It reports to Logo whatever is typed on the keyboard. Try typing

```
READCHARACTER
```

or

```
RC
```

Nothing happens. Next press the number 9. Logo responds with

```
Result:  9
```

In the last chapter, you learned that each Logo instruction needs to begin with a command. READCHARACTER is a reporter. When you type READCHARACTER all by itself on a line, Logo first waits for a character to be typed and then informs you that it doesn't know what to do now.

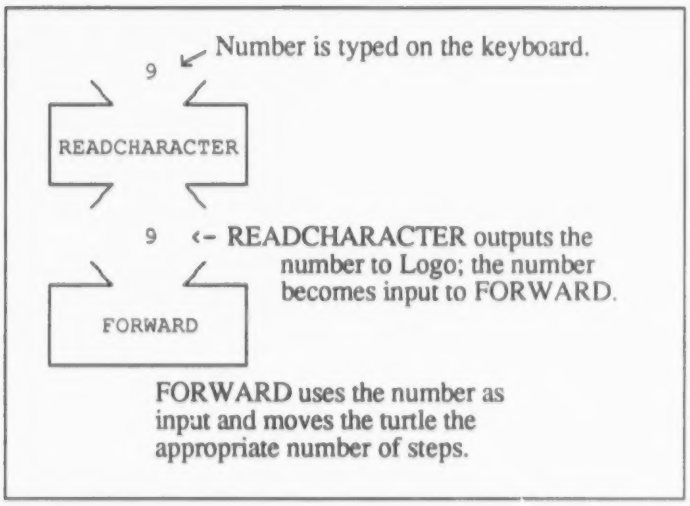
Next try typing:

```
FORWARD READCHARACTER
```

or

```
FORWARD RC
```

Again, type the number 9. The turtle moves forward 9 steps. READCHARACTER *reported* the number 9 to FORWARD. FORWARD then used 9 as input. Logo responded by moving the turtle 9 turtle steps. (Using the ideas from the last chapter, recall that FORWARD is a *command* and the *effect* caused by FORWARD is to move the turtle.)


Number is typed on the keyboard.
9
READCHARACTER
9 <- READCHARACTER outputs the number to Logo; the number becomes input to FORWARD.
FORWARD
FORWARD uses the number as input and moves the turtle the appropriate number of steps.

Take a few minutes to reread the above discussion. It refers to many of the new terms given in the last chapter. Perhaps you might even want to go back to that chapter and read the definitions of reporter and command again.

If you know another computer language such as BASIC or Pascal, you are more likely to find interactive reporters confusing. In both BASIC and Pascal, you have a variable into which you put any value typed on the keyboard. This value is then available by using the variable. (More information about such ideas in Logo is given in Chapters 26 and 35.) READCHARACTER is *not* a variable. It does *not* hold the value for later use. Instead, it immediately reports the value to Logo. The action of READCHARACTER can be summarized as follows:

1. Logo waits until a key on the keyboard is pressed.
2. The *first* key pressed is reported by READCHARACTER *immediately*.
3. If READCHARACTER has a command in front of it, that command tries to use the character reported.
   If there is no command in front of READCHARACTER, it reports to Logo; Logo responds with `"Result: ---"`

Next, look at the use of READCHARACTER in the procedure

```
TO SEE.TURTLE
TELL READCHARACTER
SHOWTURTLE
FORWARD 50
END
```

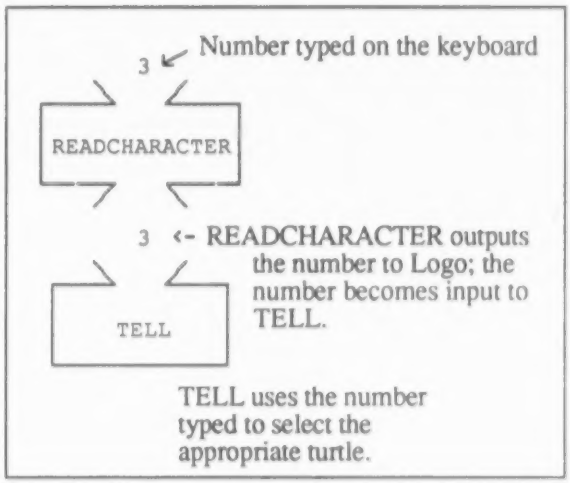
Type

```
SEE.TURTLE
```

Then type a number between 1 and 9. One of the turtles appears and moves forward 50 turtle steps. Type

```
CLEARGRAPHICS
HIDETURTLE
SEE.TURTLE
```

and try a different number.


3
Number typed on the keyboard
READCHARACTER
3
<- READCHARACTER outputs the number to Logo; the number becomes input to TELL.
TELL
TELL uses the number typed to select the appropriate turtle.

Do you see that you can "talk to" any turtle you want by using this procedure? In this case, READCHARACTER is reporting the number typed to the command TELL. TELL then uses the number to "activate" the appropriate turtle. When Logo encounters the SHOWTURTLE command, the active turtle appears.

You can also use READCHARACTER to change the mode used by the pen, for example,

```
TO PICK.MODE
PRINT [WHAT PEN MODE DO YOU WANT?]
SETPENMODE READCHARACTER
END
```

Or, to pick a pen pattern, type

```
TO PICK.PATTERN
PRINT [WHAT PATTERN DO YOU WANT?]
SETPPATTERN READCHARACTER
END
```

Note that you can only use pen patterns from 0 to 9. If you want to be able to select patterns from, say, 10 to 19, you can use

```
TO PICK.PATTERN
PRINT [WHAT PATTERN DO YOU WANT?]
SETPPATTERN 10 + READCHARACTER
END
```

Here 10 is added to the number you type. However, with this second version of the procedure, you can't use pen patterns 0 to 9. You will learn how to enter more than one number or letter at a time later in this book.

Here's a more elaborate example:

```
TO MAKE.SQS
SET.UP
WHICH.TURTLE
SQUARE
END

TO SET.UP
CLEARGRAPHICS
TELL [0 1 2 3 4 5 6 7 8 9]
HOME
HIDETURTLE
END

TO WHICH.TURTLE
PRINT [WHICH TURTLE DO YOU WANT? ]
TELL READCHARACTER
SHOWTURTLE
END

TO SQUARE
REPEAT 4 [FORWARD 50 RIGHT 90]
END
```

When you type

```
MAKE.SQS
```

the message

```
WHICH TURTLE DO YOU WANT?
```

appears and you can then type a number from 0 to 9 for the turtle that you want to draw a square. As before, READCHARACTER reports to TELL. TELL activates the turtle. (The *effect* of TELL is to activate a turtle.) Then the turtle appears. Finally, the active turtle follows the directions in the procedure SQUARE.

Notice the SET.UP procedure in this program. It makes sure that all the turtles are in the center of the Turtle Window and hidden before you begin. It is a good idea to include a procedure to make sure the Turtle Window or Logo Window is the way you want it before you start a program. Sometimes that means using just CLEARGRAPHICS; at other times it means worrying about the placement of a number of turtles. At still other times it means including commands to set the pen size and pen pattern.

Later you will learn other ways to write interactive programs and other ways to use READCHARACTER. You should take some time now to be sure you see how READCHARACTER works. While you are practicing, verbalize the ideas about commands and reporters used in this chapter. As you do so, your understanding of Logo grammar will increase.

Tips and Techniques

Keep in mind that you will *never* have READCHARACTER on a line by itself. Every Logo instruction must begin with a command, and READCHARACTER is a reporter. READCHARACTER must report – send – a value to a command.

You can only read one character at a time with READCHARACTER. When using READCHARACTER don't try to read numbers with more than one digit or LOGO words containing more than one character. Later you will learn reporters that read words or lists.

READCHARACTER does not save the value you type. Think of READCHARACTER as a "funnel" between the keyboard and a Logo command. If you use a funnel to pour liquid from one container to another, there is no liquid left in the funnel when you are done. If you use READCHARACTER to move a character from the keyboard to Logo, there are no characters "left in READCHARACTER."

Project Suggestions

Write a multiple turtle program of your own that allows the user to tell which turtle s/he wants to use. You can have the turtle make elaborate designs using different pen patterns, pen sizes, and pen modes.

Write a program that asks the user what pen pattern, size, or mode s/he wants to use before drawing a design.

Select any procedure you have used that includes READCHARACTER. See if you can draw a diagram like the ones in this chapter to picture the action taking place in your procedure.

19. Indefinite Repetition

Whenever you want to repeat some action in Logo, you use the REPEAT command. REPEAT is followed by the exact number of times you want the action to happen. So, if you want to draw a figure with six sides, a 6 is placed after the REPEAT. You must know exactly how many times you want an action to occur when you use REPEAT. This is sometimes called a definite loop.

Looping is a powerful idea in computing. Computers can do repetitive tasks endlessly without tiring. However, sometimes you do not know exactly how often you want an action to occur. For example, you might not know exactly how many people turned in ballots to be tallied. Or, you might not know how many people got overtime this pay period. You might have a design that you want to appear in the Turtle Window until the user stops the program. There are many examples in real life of *indefinite* loops.

In this chapter, you will take the first step toward learning to construct loops in Logo that can be used to repeat some action an unpredictable number of times. This is accomplished using an idea called *recursion*. To begin, you will learn to create loops that go on forever.

Suppose you have the following procedure:

```
TO DRAW.IT
FORWARD 50
RIGHT 177
END
```

If you type

```
DRAW.IT
```

and then move the cursor up and run DRAW.IT again a number of times, a "fluffy ball" design begins to form:

You could write another procedure

```
TO DESIGN
REPEAT 50 [DRAW.IT]
END
```

to create the fluffy ball. It is interesting to watch the patterns form as the lines draw over each other. Perhaps you would like this effect to continue an indefinite number of times.

Modify the procedure in the Edit Window as follows:

```
TO DRAW.IT
FORWARD 50
RIGHT 177
DRAW.IT
END
```

Now when you type

```
DRAW.IT
```

the effect goes on and on. Use Command-G (⌘-G) to stop the procedure DRAW.IT. Do you see what is happening?

1. Procedure DRAW.IT is called.

2. The turtle goes forward 50 turtle steps.

3. The turtle turns right 177 degrees.

4. Procedure DRAW.IT is called — but this is the same as step 1!

Thus, procedure DRAW.IT calls procedure DRAW.IT indefinitely! Procedure DRAW.IT is said to be *recursive*. The command DRAW.IT just before the word END is the *recursive call* to the procedure DRAW.IT.

Here's another idea to try. Suppose you want to leave a message on the computer screen as a reminder to a friend. You could write a recursive procedure to accomplish this task:

```
TO REMIND
PRINT [DON'T FORGET THE DANCE ON SATURDAY!]
PRINT [
WAIT 20
REMIND
END
```

Recall that WAIT 60 causes Logo to pause for one second. Now, if you type REMIND, the message will continue to appear over and over until you press Command-G.

Infinite looping is fun to use to get flashy effects–but it's not particularly useful in and of itself. The fact that a procedure can call itself is a powerful idea. Not all computer languages allow recursion; for example, BASIC doesn't. You will learn how to use recursion in a more useful framework later in this book.

One word of warning is in order at this point. For now, when you write recursive procedures, be sure that recursive calls are always the last line before END. This is called *tail recursion*. Otherwise, Logo is never able to get to the end of your procedure, and you may get some very unexpected results, for example

```
TO OOPS.DRAW
FORWARD 50
OOPS.DRAW
RIGHT 177
END
```

If you run this procedure, you will no longer get a fluffy ball. This procedure functions as follows:

1. Procedure OOPS.DRAW is called.
2. The turtle goes forward 50 turtle steps.
3. Procedure OOPS.DRAW is called.

Logo is never able to get to the RIGHT 177. The procedure does not run as you expect. This placement of OOPS.DRAW before the end of the procedure OOPS.DRAW is called *embedded recursion.* You will learn more about it later. In the meantime, keep in mind that your recursive procedure calls should always be just before the END of the procedure.

Tips and Techniques

The important idea in this chapter is looping. For now the infinite loops that you can write using recursion are not really useful. It is better not to try using them in programs at this point. Instead, simply practice the idea so that you understand that a procedure can call itself.

Embedded recursion can cause many bugs in programs. Be very careful when you start using recursive calls in programs. Make sure the recursive call is on the last line of the procedure before the END. Nothing should be after the recursive call if you are using tail recursion to build loops.

You can build loops that use several procedures, for example,

```
TO DRAW.LINES
FORWARD 50
RIGHT 177
CHANGE.PP
END

TO CHANGE.PP
SETPPATTERN RANDOM 39
DRAW.LINES
END
```

DRAW.LINES is still a tail recursive procedure call. Most likely you would write the above as one procedure. However, if your procedures get very long or complex, you may want to subdivide them as in the above example. Then the recursive call to the "first" procedure should be on the last line of the "last" procedure.

Project Suggestions

Create a procedure that has an interesting effect. You might display a flashy graphic design, draw with random pen patterns, or create some mixture of graphics and text. Turn your procedure into a recursive procedure that continues indefinitely. Next, make your procedure part of a larger program, but make sure that your recursive procedure is the last procedure you use in your program. Otherwise, you will never reach the end of your program. (You'll need to use Command-G to stop your recursive procedure in any case.)

20. Animation

No doubt you have seen computer programs that have objects moving across the screen. If you have played video games, you often see objects, characters, or space ships in constant movement. Using what you know about making shapes and about recursion, you can write procedures to cause movement on the screen.

Did you save the helicopter shape that was created in Chapter 5? If so, load the file. If not, you may want to go back to Chapter 5 and recreate that shape. Now type

```
SHOWTURTLE
SETSHAPE :HELICOPTER
PENUP
RIGHT 90
FORWARD 2
```

Use the arrow keys or mouse and the Enter key to repeat the last command a number of times. The helicopter moves slowly towards the side of the Turtle Window. You can put this in a program that includes a recursive procedure:

```
TO FLY
SET.UP
GO.FORWARD
END

TO SET.UP
CLEARGRAPHICS
SETSHAPE :HELICOPTER
SHOWTURTLE
PENUP
RIGHT 90
END

TO GO.FORWARD
FORWARD 2
WAIT 2 <—This instruction keeps the helicopter from moving too fast.
GO.FORWARD
END
```

Now try the FLY procedure. The helicopter moves to the right. You can easily change the speed at which the helicopter moves. Make the number after WAIT larger. Make it smaller. (You can use decimals.) If you just use FORWARD by itself, you have less control over the speed. Take some time to experiment with these ideas. You should become comfortable changing the inputs to both WAIT and FORWARD to adjust the speed of animation. Use Command-G (⌘-G) to stop the movement.

The movement of the helicopter involves only one shape. Animation of the sort you see in cartoons and in many computer programs often involves movement of parts of the shape itself as well. You can do this in Logo by creating several shapes.

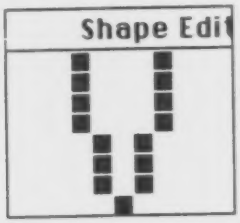
First, load the Shape Edit file from disk and then type

```
SHAPE.EDIT
```

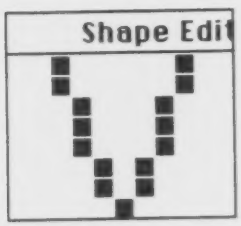
Next, enter

```
11 8
```

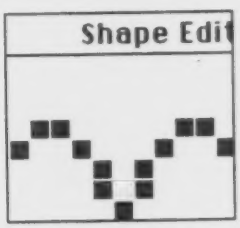
for the size of the shape. Use your mouse to create this shape:



Type "N" and name your shape BIRD.1 (Don't worry about pen location or dimensions for now.) You can either change the shape you see on the screen, or type "C" and create a new shape like this:



Name this shape BIRD.2. Repeat the above process to get the shape BIRD.3 as shown below:



Now go to the Editor and enter this procedure:

```
TO FLAP
SETSHAPE :BIRD.1
WAIT 10
SETSHAPE :BIRD.2
WAIT 10
SETSHAPE :BIRD.3
WAIT 10
SETSHAPE :BIRD.2
WAIT 10
SETSHAPE :BIRD.1
WAIT 10
FLAP
END
```


Now type

```
FLAP
```

Do you want the bird to flap its wings faster or slower? If so, you can change the numbers after the WAIT commands.

To make the bird move as well as flap its wings, add some FORWARD commands. However, adding FORWARD commands between each SETSHAPE command will make the procedure quite long. Instead, divide the procedure into two procedures—one to change the turtle shape and one to move the turtle:

```
TO FLAP
SETSHAPE :BIRD.1
MOVE
SETSHAPE :BIRD.2
MOVE
SETSHAPE :BIRD.3
MOVE
SETSHAPE :BIRD.2
MOVE
SETSHAPE :BIRD.1
MOVE
FLAP
END

TO MOVE
FORWARD 1
WAIT 10
END
```

Now type

```
PENUP
RIGHT 90
FLAP
```

Notice that the commands that control the speed are in the subprocedure MOVE. To make the bird fly faster or slower, you can change the numbers after the WAIT or the FORWARD commands in procedure MOVE.

Then you can make the bird appear to fly away by writing this procedure:

```
TO FLY.AWAY
REPEAT 20 [FLAP]
END
```

If you want the flying to continue indefinitely, then you can write this procedure:

```
TO FLY.FOREVER
FLAP
FLY.FOREVER
END
```

Since you can use more than one turtle in *Terrapin Logo*, you can have two shapes moving at once. This, however, is a bit more complex than you might expect. Keep in mind that the computer can do only one thing at a time. To animate two different shapes at once doing different things requires some planning and careful programming.

Begin by looking at a simple example. Suppose you would like to have Turtle 0 fly away as a bird while Turtle 1 flies off as a helicopter. First you want Turtle 0 to change its shape and move a bit. Then you want Turtle 1 to move a bit. Your solution to this problem might look like this:

```
TO ALL.FLY
TELL 0
SETSHAPE :BIRD.1
MOVE
TELL 1
SETSHAPE :HELICOPTER
MOVE.HELI
TELL 0
SETSHAPE :BIRD.2
MOVE
TELL 1
SETSHAPE :HELICOPTER
MOVE.HELI
TELL 0
SETSHAPE :BIRD.3
MOVE
TELL 1
SETSHAPE :HELICOPTER
MOVE.HELI
TELL 0
SETSHAPE :BIRD.2
MOVE
TELL 1
SETSHAPE :HELICOPTER
MOVE.HELI
TELL 0
SETSHAPE :BIRD.1
MOVE
TELL 1
SETSHAPE :HELICOPTER
MOVE.HELI
TELL 0
ALL.FLY
END
```

```
TO MOVE
FORWARD 1
WAIT 10
END

TO MOVE.HELI
FORWARD 5
WAIT 10
END
```

But this ALL.FLY procedure is quite long. We can move a number of commands into the MOVE.HELI procedure:

```
TO ALL.FLY
TELL 0
SETSHAPE :BIRD.1
MOVE
MOVE.HELI
SETSHAPE :BIRD.2
MOVE
MOVE.HELI
SETSHAPE :BIRD.3
MOVE
MOVE.HELI
SETSHAPE :BIRD.2
MOVE
MOVE.HELI
SETSHAPE :BIRD.1
MOVE
MOVE.HELI
ALL.FLY
END

TO MOVE
FORWARD 1
WAIT 10
END

TO MOVE.HELI
TELL 1
SETSHAPE :HELICOPTER
FORWARD 5
WAIT 10
TELL 0
END
```

As you can see, you must be very careful when writing procedures such at this. It is easy to create very long programs that are difficult to debug. Remember to break your procedures into small, meaningful parts. When you have learned more Logo programming, you will see that procedures like ALL.FLY can be written much more efficiently.

Using the methods described above, you can create a wide variety of interesting movement. Animation is a lot of fun and can add interest to any project. With the ability to create the shapes of your choice, your projects are limited only by your own creativity.

Tips and Techniques

By now you are writing more complex programs. As mentioned above, you should always keep your procedures short and give them meaningful names. This makes debugging easier.

Another helpful hint for debugging is to make sure each procedure works by itself. Don't always type the name of the top level procedure and run the entire program. Run each procedure and make sure it works. Run the entire program only when you are sure all the parts work.

When you have a lot of procedures, it is easy to lose track of the action of each procedure. One way to keep track of which procedure is running is to put PRINT instructions in each procedure. For example,

```
TO FLY.AWAY
PRINT [IN PROCEDURE FLY.AWAY]
REPEAT 20 [FLAP]
END
```

prints the words "IN PROCEDURE FLY.AWAY" each time the FLY.AWAY procedure is called. This gives you a trace of the action of the program. Putting in PRINT instructions will slow down the action of your program and clutter your screen. However, it will help you see what is happening as your program runs.

Logo also contains some useful primitives to help you debug. First, open the Debug Window from the Menu bar (or type DEBUG in the Logo window), and then type

```
TRACE [FLY.AWAY FLAP]
```

and then run FLY.AWAY. Logo will print messages telling you what part of your program is being run. You can turn off TRACE by typing

```
UNTRACE [FLY.AWAY FLAP]
```

Of particular use when working with animation is the ability to temporarily stop your procedure. If you press Command-P (⌘-P), your program will pause until you type CONTINUE (CO). There is more information on debugging commands and reporters in the Appendices.

Project Suggestions

Create a street scene with houses and trees and have a car driving down the street.

Draw a picture of an airport and have an airplane flying by, or perhaps a helicopter landing...or both!

Create a more elaborate animation of a bird flying. Be careful to break your animation into procedures of reasonable length.

Define a number of shapes and create your own animated movie. Be careful to plan your program carefully so that none of your procedures gets too long!

21. Creating Definite Loops

So far when you have used recursive calls to repeat an action, you needed to use Command-G to interrupt the program. This is not very useful if you want the program to stop automatically or if you want to have the user ask it to stop. To stop recursion, you need to learn about the IF primitive. IF is a Logo primitive that allows you to make decisions based on some condition.

Begin by typing

```
CLEARGRAPHICS
PRINT DOT?
```

You see

```
FALSE
```

DOT? is a reporter that returns TRUE if the turtle is over a black dot (or pixel) and FALSE if it is over a white dot (or pixel). If you type

```
PENDOWN
FORWARD 10
PRINT DOT?
```

Logo returns TRUE since the pen "point" is over the end of the line that the turtle just drew.

Begin by writing a procedure to fill a square with black:

```
TO BLACK.SQ
CLEARGRAPHICS
PENDOWN
SETPPATTERN 1
FILLSH [REPEAT 4 [FORWARD 50 RIGHT 90]
END
```

Next, write a procedure to place the turtle below the square:

```
TO PLACE.TURTLE
PENUP
BACK 15
END
```

Now send the turtle on a "random walk":

```
TO RANDOM.WALK
RIGHT RANDOM 30
FORWARD 5
PRINT DOT?
RANDOM.WALK
END
```

Most likely you see the turtle move in and out of the black square in a circular pattern. As you watch the Logo Window, you first see FALSE a number of times, and then TRUE several times as the turtle travels over the black square. The motion of the turtle continues until you press Command-G.

Suppose you wanted the turtle to move until it landed on the black square. You could modify RANDOM.WALK as follows:

```
TO RANDOM.WALK
RIGHT RANDOM 30
FORWARD 5
IF (DOT? = "FALSE) [RANDOM.WALK]
END
```

The condition after IF is "(DOT? = "FALSE)." You can then read this instruction "if there is not a dot under the turtle, call the RANDOM.WALK procedure."

Examine the IF instruction more carefully. Notice that it is made of three parts:

1. the word IF
2. a condition, in this case (DOT? = "FALSE)
3. a list of commands (enclosed in square brackets), in this case only RANDOM.WALK.

Logo checks to see if the condition (DOT? = "FALSE) is true. If it is true, it runs the list of commands "[RANDOM.WALK]."

When you run RANDOM.WALK, you see that the turtle continues to move until it lands on black. Sometimes the turtle will move quite a lot; at other times, it will not move much at all. The amount of movement depends on the number reported by RANDOM for the amount of turn.

Here's another way to stop a loop. Suppose you want to ask the user a question, such as "Do you want to see the design again?" The following procedure will do just that:

```
TO ASK.QUES
DESIGN  <— This a procedure you have defined elsewhere.
PRINT [DO YOU WANT TO SEE THE DESIGN AGAIN?]
IF READCHARACTER = "Y [ASK.QUES]
END
```

The three parts of this IF statement are:

- the word IF
- the condition: READCHARACTER = "Y
- the action to take if the condition is true, namely, call ASK.QUES.

Thus, this procedure

1. runs a procedure called DESIGN, which has been defined previously.
2. prints a message in the Logo Window: "DO YOU WANT TO SEE THE DESIGN AGAIN?"
3. checks to see if the character reported by READCHARACTER is an uppercase Y.
4. calls the procedure ASK.QUES if the character is an uppercase Y.
5. goes to the END and stops if the character is not an uppercase Y.

Notice that the character Y is preceded by a quotation mark ("). Without the quotation mark, Logo would look for a procedure called Y. READCHARACTER reports a word made up of one character. You must *quote* the Y for Logo to compare what READCHARACTER reports with the letter Y.

You will learn more about using IF later. For now, you can begin to use it to stop simple recursive procedures.

Tips and Techniques

When the condition after an IF instruction is true, the list in the IF instruction is run. When the condition is false, the next instruction in the procedure is run. Later you will learn how to have two different actions happen in the IF instruction: one when the condition is true, another when it is false.

Keep in mind that there are three parts to the IF instruction:

- In the first part, the word IF has no punctuation
- The condition is the second part and may contain some punctuation, such as parentheses
- The third part is the list of actions to run.

```
IF (DOT? = "TRUE) THEN PRINT [THE BACKGROUND IS BLACK.]
```

A condition in an IF must return true or false. One way to see if your condition is functioning correctly is to run it by itself. For example, if you type

```
DOT? = "TRUE
```

then Logo responds

```
RESULT: TRUE
```

or

```
RESULT: FALSE
```

If the message on your screen is anything else, then you have written your condition incorrectly.

There is no "does not equal" sign in Logo. If you wanted to say "If DOT? is not true, call the procedure RANDOM.WALK" you could write

```
IF NOT (DOT? = "FALSE) [RANDOM.WALK]
```

Simply put the word NOT in front the condition.

When using standard math symbols such as >, <, and = in conditions, it is a good idea to use parentheses. This has to do with the way Logo "parses" or translates the words you type so the computer can understand them.

Inserting PRINT into procedures like the one at the beginning of this chapter can be very helpful in debugging.

```
TO RANDOM.WALK
RIGHT RANDOM 30
FORWARD 5
PRINT [MOVED 5 TURTLE STEPS]
IF (DOT? = "FALSE) [RANDOM.WALK]
END
```

tells you how far the turtle has moved before the condition is tested in the IF instruction. Because this happens so fast, you might also want to insert a WAIT command:

```
TO RANDOM.WALK
RIGHT RANDOM 30
FORWARD 5
PRINT [MOVED 5 TURTLE STEPS]
WAIT 20
IF (DOT? = "FALSE) [RANDOM.WALK]
END
```

Be sure to put the PRINT command before the recursive call.

You may have noticed that DOT? is a special kind of reporter. It can return only the values TRUE and FALSE to Logo. A reporter that gives only TRUE and FALSE as results is called a *predicate*. You will learn more about predicates in Chapter 37.

Project Suggestions

Write a program that displays a message in the Logo Window over and over until the user types something other than Y.

Write a program that randomly picks a pen pattern and draws a filled square with that pattern at a random spot in the Turtle Window until pen pattern 0 is picked.

Write a simple game program that sends the turtle on a random walk on a screen with a variety of dark areas. In fact, you might even experiment with using different pen patterns. Can the turtle "navigate" across some of them successfully?

22. Stamping Areas

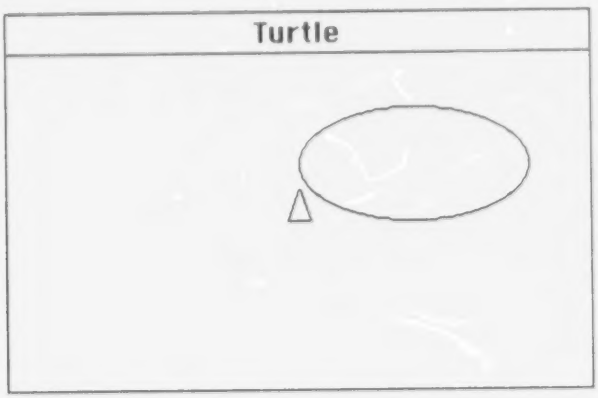
Terrapin Logo has a number of features that look quite familiar to users of programs such as MacPaint. In this chapter you will learn how to stamp arcs, rectangles, and ovals in the Turtle Window by using a single command. In addition, when you stamp one of these shapes, you can fill the area with the current pen pattern, still using only one command.

Stamping Ovals

You can stamp an egg-shaped area in the Turtle Window using the STAMPOVAL command. This commands accepts the width and height of the rectangle into which the oval will fit. Thus, if you type

```
STAMPOVAL 100 50
```

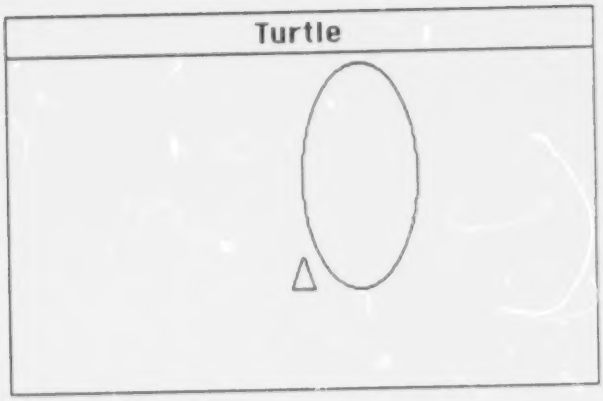
you see



If you type

```
CLEARGRAPHICS
PENUP
BACK 30
PENDOWN
STAMPOVAL 50 100
```

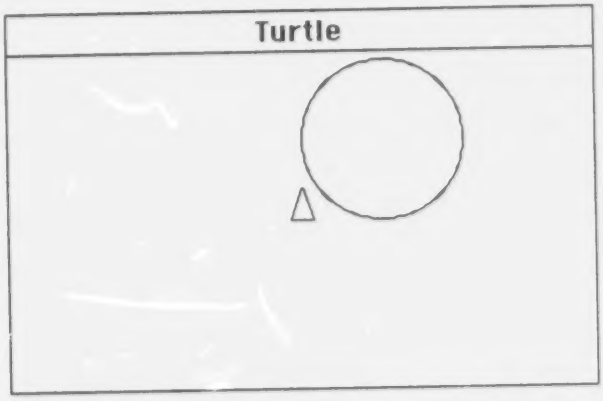
you see



And if you type

```
STAMPOVAL 70 70
```

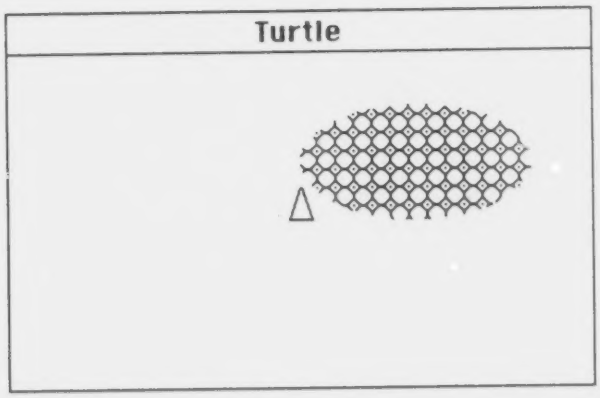
you see



You can also fill an oval with the current pen pattern:

```
SETPPATTERN 15
(STAMPOVAL 100 50 "TRUE)
```

Then you see



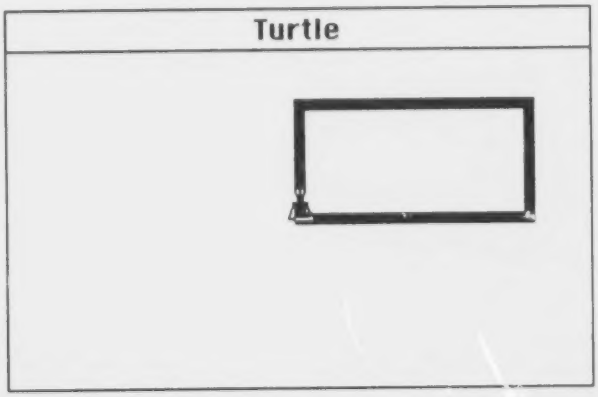
Note that to fill an oval with the current pen pattern, you must add the word "TRUE" at the end of the command, you must put a quotation mark (") before TRUE in order to quote it, and you must surround the entire command with parentheses. Technically, the third input to STAMPOVAL (the TRUE) is a predicate. You will learn more about predicates later in this book. STAMPOVAL is another example of a command that will accept a variable number of inputs. Without parentheses, it will accept only two inputs. If you use parentheses, then it will accept either two or three.

Stamping Rectangles

As you might guess, stamping rectangles is much like stamping ovals. Thus, if you type

```
SETPENSIZE 5 5
STAMPRECT 100 50
```

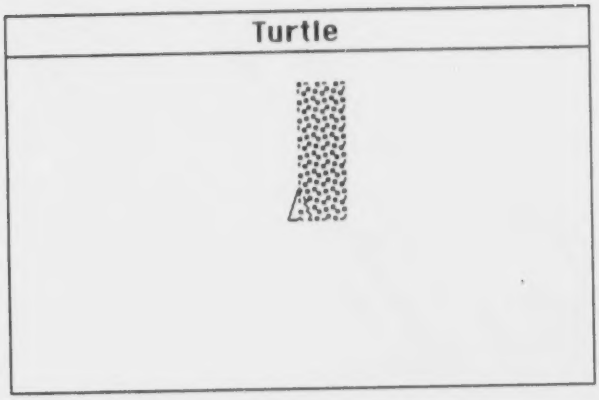
you see



and if you type

```
SETPENSIZE 1 1
SETPPATTERN 9
(STAMPRECT 20 60 "TRUE)
```

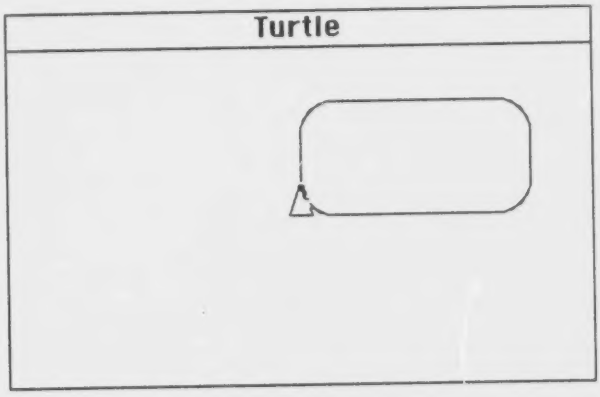
you see



The STAMPRECT command draws rectangles with square corners. You can also stamp rectangles with rounded corners using STAMPRRECT. So, if you type

```
STAMPRRECT 100 50 30
```

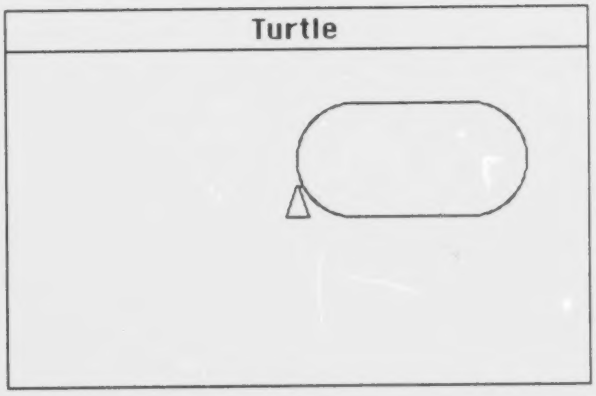
you see



And if you type

```
STAMPRRECT 100 50 50
```

you see

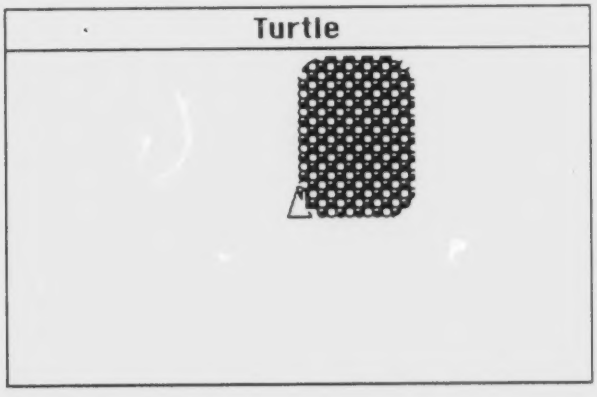


Notice that the third input to STAMPRRECT is the radius of the curve in each corner in turtle steps. To understand how the corners are drawn, think of the turtle as moving inside the rectangular area. Then the turtle positions itself to draw a circle with the given radius so that the arc of the circle meets the sides of the rectangle.

Of course, you can fill a rounded rectangle with a pen pattern as well:

```
SETPPATTERN 37
(STAMPRRECT 50 70 25 "TRUE)
```

You see



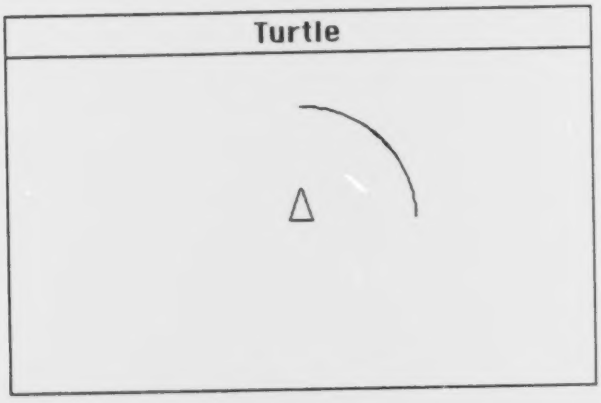
Stamping Arcs

In addition to the closed figures you stamped above, you can also stamp curved lines, called arcs. To stamp an arc you supply an x-radius, a y-radius, and a number of degrees. Several examples may help you understand these three inputs.

Begin by typing

```
STAMPARC 50 50 90
```

You see one-fourth of a circle that has a radius of 50 turtle steps:

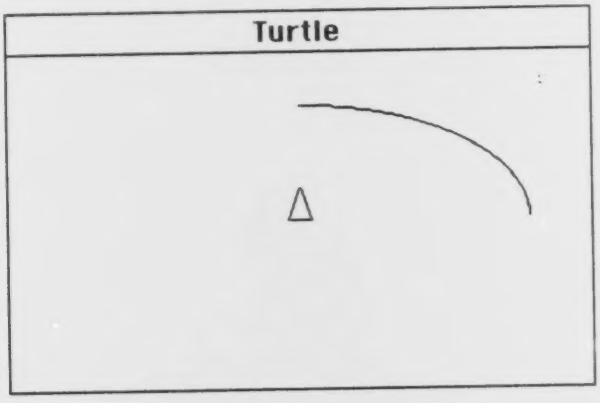


To get half a circle, you use 180 for the number of degrees; for a full circle, you use 360.

If you use different numbers for the x- and y-radii, then you get a piece of an oval (ellipse). For example, if you type

```
STAMPARC 100 50 90
```

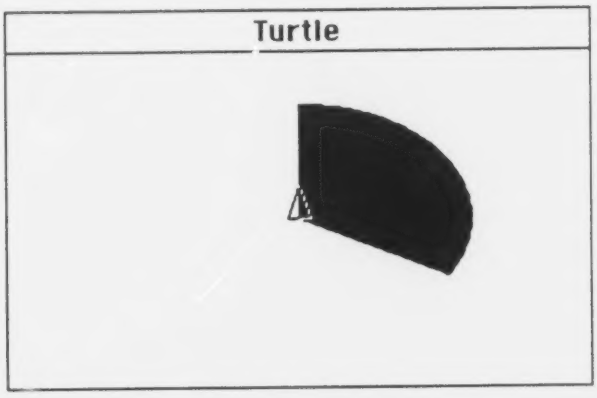
you see



Since arcs are not closed figures, you may be surprised to learn that you can fill them as well! So, if you type

```
SETPPATTERN 27
(STAMPARC 75 50 120 "TRUE)
```

you see



Notice that STAMPARC can be used to create part of a filled oval.

Tips and Techniques

Always be sure the pen is down when you are stamping shapes. Sometimes you may think you have done something wrong when the only problem is that the pen is up.

Have you noticed that when you stamp a rectangle, oval, or arc that is filled with the current pen pattern, there is no border? If you want a shape with a border, you must use a series of commands. First, draw a border of the width you want:

```
CLEARGRAPHICS
SETPPATTERN 1
SETPENSIZE 5 5
STAMPRECT 100 50
```

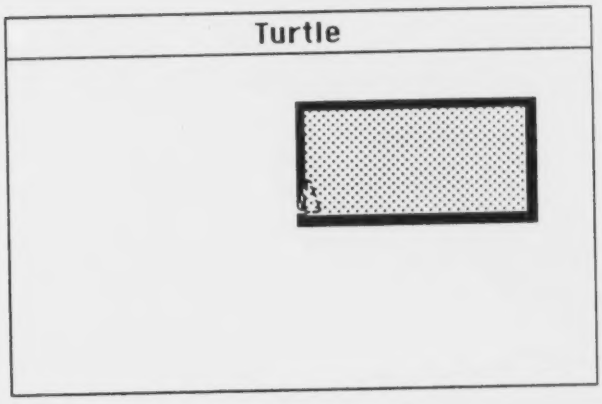
Next, move the turtle to the corner of the inside of your border:

```
PENUP
FORWARD 2.5
RIGHT 90
FORWARD 2.5
LEFT 90
PENDOWN
```

Notice that the turtle was moved half the width of the pen. Finally, stamp a rectangle that will fit inside of the border:

```
SETPPATTERN 22
(STAMPRECT 95 45 "TRUE)
```

You see

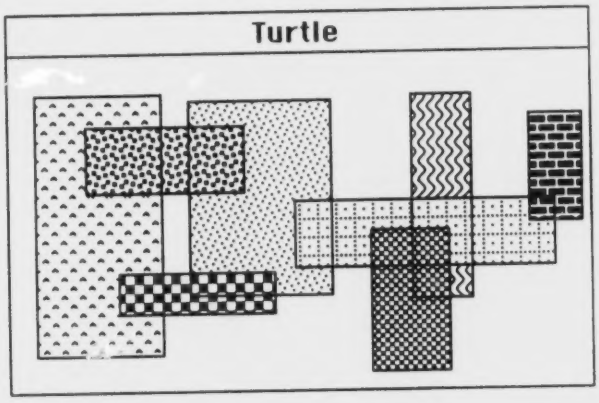


Project Suggestions

These commands offer a lot of possibilities, but they need a number of rather complex inputs. Spend some time experimenting with different inputs to each one until you understand how each shape-stamping command works.

Can you make a picture that shows some figures and their shadows? For example, draw a ball with a long, oval-shaped shadow as if the sun were low on the horizon.

Create a Mondrian-like painting with a lot of rectangles filled with different patterns:



Can you use RANDOM with these commands in interesting ways so that you can get "modern art" in the Turtle Window?

23. More Interactive Programming

You have learned how to use READCHARACTER to accept a single character from the keyboard. With it, the user can interact with the program. For example,

```
TO PICK.TURTLE
PRINT [WHAT TURTLE NUMBER DO YOU WANT?]
TELL READCHARACTER
SHOWTURTLE
END
```

allows the program user to select the turtle to use.

Logo has another primitive, called READLIST (RL), for writing interactive programs. READLIST allows you to read more than one character at a time from the screen. READLIST is a reporter that returns a list typed at the keyboard. When you use READCHARACTER, the key you press is accepted immediately. You must press the Enter key to have READLIST accept information from the keyboard.

Here is an example:

```
TO GREET
PRINT [WHAT IS YOUR NAME]
(PRINT [GLAD TO MEET YOU, ] READLIST)
PRINT []
PRINT [WHAT IS YOUR FAVORITE COLOR?]
(PRINT [I LIKE] READLIST [TOO!])
END
```

If you type

```
GREET
```

then

```
WHAT IS YOUR NAME?
```

appears in the Logo Window. If you type

```
GEORGE
```

then you see

```
GLAD TO MEET YOU, GEORGE

WHAT IS YOUR FAVORITE COLOR?
```

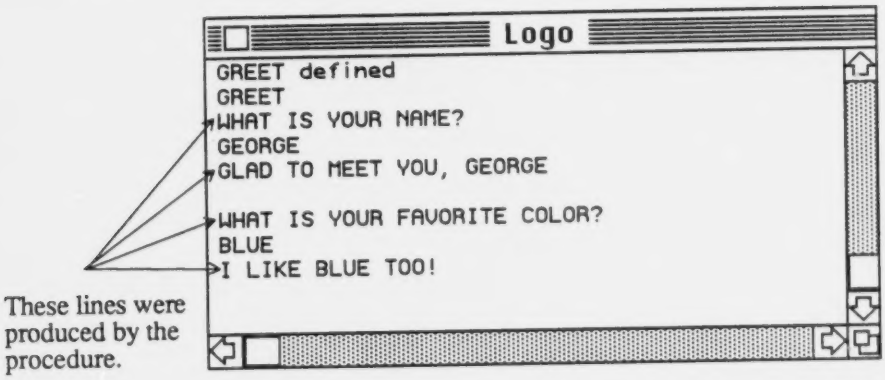
Suppose you type

```
BLUE
```

Then

```
I LIKE BLUE TOO!
```

appears in the window.



Using READLIST allows you to accept the user's responses of almost any kind from the keyboard.

Creating READWORD Procedures

So far you have learned how to accept both single characters and lists entered by the user at the keyboard. You used READCHARACTER to accept single letters or numbers. You used READLIST to accept lists of words or numbers. However, sometimes you want to use just a single Logo word or a number. Recall that a number is a special kind of word.

If you want to work with Logo words of more than one character, you can't use either READCHARACTER or READLIST. There are several ways to solve this problem. Perhaps the simplest is to write a couple of special procedures and save them in a file that you can add to your workspace whenever you need them.

Clear your workspace with ERALL. Go to the Editor and enter these procedures:

```
TO READWORD
OUTPUT FIRST READLIST
END
```

Save this procedure in a file, perhaps as READWORD. Then, when you want to use this procedure in another program, you can type

```
LOAD "READWORD
```

The procedures in the READWORD file will be added to your workspace. Don't worry about the "inner workings" of the READWORD for now. You will learn about OUTPUT in Chapter 31. FIRST will be discussed in the chapters on traditional Logo list processing at the end of this book.

Using these procedures, you can now work with words and numbers. Here, for example, is a procedure for drawing designs interactively:

```
TO DRAW.IT
PRINT [HOW FAR FORWARD?]
FORWARD READWORD
PRINT [HOW MUCH RIGHT TURN?]
RIGHT READWORD
DRAW.IT
END
```

If you type DRAW.IT, you see

```
HOW FAR FORWARD
```

If you type

```
50
```

the turtle will move forward 50 steps. Then you see

```
HOW MUCH RIGHT TURN?
```

If you type

```
90
```

the turtle will turn right. Then the questions are repeated over and over until you press Command-G. This procedure makes a nice drawing utility for beginners.

Using READWORD, you could create a very simple guessing game, for example,

```
TO GUESS
PRINT []
PRINT [GUESS A NUMBER:]
IF NOT (READWORD = 21) [GUESS]
END
```

When you type GUESS, the program asks you to guess a number. It continues to ask you to guess until you type 21. Notice the IF instruction. The condition after IF is

```
NOT (READWORD = 21)
```

Logo checks to see if the number typed is equal to 21. If it is, then (READWORD = 21) is equal to TRUE. The NOT reverses the value to FALSE, and Logo moves to the next line of the procedure, encounters END, and stops. If READWORD is not 21, then (READWORD = 21) is equal to FALSE and the NOT reverses the value to TRUE and the procedure GUESS is called again.

Tips and Techniques

You already know how READCHARACTER works. Don't become overwhelmed by READLIST and READWORD. There is very little difference between READCHARACTER and the new reporters in this chapter. They are only "funnels" between the keyboard and Logo.

When you use READCHARACTER and READLIST, it is impossible to tell by looking at the screen that input is expected from the keyboard. Always be sure you "prompt" the user by asking a question when you want input, for example,

```
(PRINT [TYPE YOUR NAME: ] READLIST)
```

When providing a prompt for a user, you may be tempted to use PRINT1 since the cursor stays on the same line when it is used. However, this approach won't work with *Terrapin Logo*. For example, if you type

```
(PRINT1 [TYPE YOUR NAME: ] READLIST)
```

and then type

```
DAVID
```

then you see

```
TYPE YOUR NAME: DAVID
```

The question *and* the answer become part of the answer!

Do not worry about understanding how READWORD works at this point. It is a reporter that really should be part of Logo but was probably omitted to save memory. Use it exactly as you would READCHARACTER or READLIST. Just remember to load it into memory when you want to use it.

Project Suggestion

Take some time to experiment with these interactive reporters. Write some simple interactive programs using READLIST and READWORD. Be sure you can use each one appropriately. Also, use different sorts of prompts to tell users that you are expecting input from the keyboard. For example, you might try

```
(PRINT [?] READLIST)
```

or

```
(PRINT [-->] READCHARACTER)
```

in your procedure.

24. Working with Files

You have been saving files since your first experiences with Logo. You have saved both pictures and your workspace. You have gotten files you saved earlier and worked with them. Most of the time you have saved and loaded files in the immediate mode.

Logo is different from many other programming languages in that anything you can do in immediate mode can also be done from a procedure. First, recall that you can load a shapes file from a procedure. For example, you might have

```
TO SET.UP
ERNAME NAMES
LOAD "HELI
SETSHAPE :HELICOPTER
END
```

and a helicopter appears in the Turtle Window. When you are using a shapes file, it is always a good idea to clear any shapes that might be in memory and then load the shapes you want. The shapes are stored in the workspace as global names (see Chapter 35). ERNAME removes a list of names from the workspace. NAMES is a reporter that returns a list of all the names in the workspace.

Similarly, you can load a picture from disk from within a procedure. Suppose you have a program that starts with a title page. Perhaps you made an elaborate turtle drawing that takes a lot of time to draw. You would rather just have the graphics appear. Thus, you might have a SET.UP procedure that looks like this:

```
TO SET.UP
CLEARGRAPHICS
LOADPICT "TITLE.PAGE
END
```

The picture that is saved in TITLE.PAGE appears in the Turtle Window.

Another way to use files is to load procedures from the workspace file of the user's choice. Suppose you have three files: SQUARE, TRIANGLE, and CIRCLE. The SQUARE file contains procedures for drawing different sizes of squares. TRIANGLE contains different triangle procedures, and CIRCLE contains some circle-drawing tools. You could have a procedure to let the user select which file s/he wants:

```
TO ASK
PRINT [WHICH FILE DO YOU WANT]
PRINT [TYPE: SQUARE  TRIANGLE  or CIRCLE]
LOAD READWORD
END
```

Recall that you need to either type or load the READWORD procedure. In this case, the file of the user's choice would be loaded into memory so that s/he could use the appropriate procedures.

Just as you can load files into memory by using a procedure, so too you can save or erase files in the workspace from within a procedure. Suppose you want a user to have a tool for saving a picture s/he created. You might have a procedure like this in your workspace:

```
TO KEEP
PRINT [DO YOU WANT TO KEEP YOUR PICTURE?]
IF READCHARACTER = "Y [SAVEPICT "MY.PICT]
END
```

This procedure can be used for young children so that they don't have to type a lot.

Similarly, you can erase files from your disk from a procedure. Of course, you want to be especially careful about writing procedures that erase files. You can easily lose a lot of work that way if you aren't careful.

Tips and Techniques

Logo allows you to manipulate files and your workspace from a procedure. If you have a very long program that won't all fit into memory at once, you can write a superprocedure that loads the first part, runs it, erases the first part from memory, loads the second part, and finally runs that second part. Such a procedure might look like this:

```
TO START
LOAD "PART.1
DO.PART.1
ERALL
LOAD "PART.2
DO.PART.2
END
```

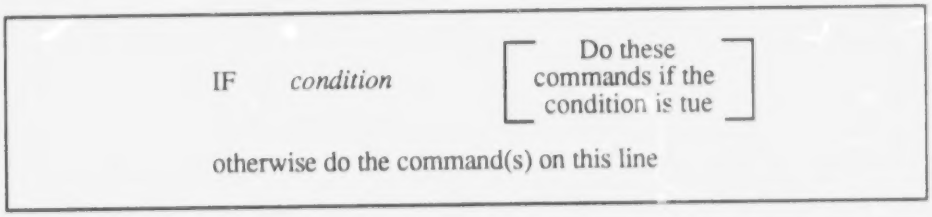
The procedure START remains in memory, controlling the entire process until the second part of the program is run.

Project Suggestions

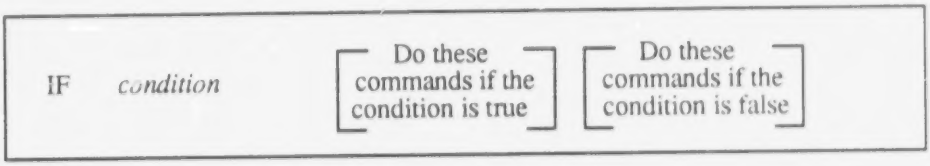
Take some time to experiment with loading and saving files from procedures. It would be a good idea to create some simple small files so that if you make any mistakes, you won't lose a lot of important work.

25. More on Making Decisions

You have used the IF instruction to write procedures that branch; that is, some decision is made within the procedure that affects the next step taken. The form of IF that you have seen takes two inputs: a condition and a list of commands.



Sometimes you want to write a procedure that causes Logo to take one action if the condition is true and another if the condition is false, and *then* you want Logo to go to the next instruction in the program. The three-input form of IF allows you to do just that:



You have seen several examples using IF to ask the user if s/he wants to repeat some action. The three-input form of IF makes such a procedure more "user-friendly":

```
TO SEE.DESIGN
DESIGN          <—This is a procedure to draw some kind of graphics.
PRINT [ANOTHER DESIGN?]
IF READCHARACTER = "Y →
      [SEE.DESIGN] →
      [PRINT [O.K.  GOODBYE, THEN!]]
END
```

In this book procedures will be formatted to make them more readable. If you type this procedure in the Editor, you do not press Return until you have typed the two brackets after "THEN!" Recall, however, that you can use the Tab key to format procedures in *Terrapin Logo*. To place "[SEE.DESIGN]" as shown above, press the Tab key after typing "Y." An arrow appears and "[SEE.DESIGN]" moves to the next line. Press the space bar several times to indent the line. Use the Tab key again to move the PRINT instruction to the next line, and use the space bar to indent.

In SEE.DESIGN, if the user types "Y," s/he sees the design again. If s/he types any other key, then the message

```
O.K.  GOODBYE, THEN!
```

appears.

Suppose you wanted to write a program to allow the user to choose from a menu, for example,

```
TO MENU
PRINT [1. SEE A PICTURE]
PRINT [2. ANSWER A QUESTION]
IF READWORD = 1 [PICTURE] [ASK.QUESTION]
END
```

When you type

```
MENU
```

you can then choose which you want to do. Notice that if the user picks 1, the procedure PICTURE is run. If the user picks anything else, ASK.QUESTION is run. If the user types the wrong key, then s/he is asked a question anyway! Later you will learn ways around this problem.

READWORD is used in the procedure MENU. Recall that you have to either write a procedure READWORD or load it from a file in order to accept numbers from the keyboard:

```
TO READWORD
OUTPUT FIRST READLIST
END
```

Tips and Techniques

In all the examples in this chapter, there is one command in each set of square brackets in the IF instruction. There can be more than one command in each set of brackets, for example.

```
IF (READWORD = 1) [FORWARD 50 RIGHT 90] [BACK 50 LEFT 90]
```

It is acceptable to put a few instructions in each list that follows IF. However, if those lists get very long, then the IF instruction gets quite hard to read. It is better to write a separate procedure for each list of commands. For example, the above instruction might read

```
IF (READWORD = 1) [GO.FD] [GO.BK]
```

where

```
TO GO.FD
FORWARD 50
RIGHT 90
END

TO GO.BK
BACK 50
LEFT 90
END
```

Keep in mind that Logo procedures are run one line at a time. That is, when Logo is running one line of a procedure, it does not "know" what the next line of a procedure will be. When you are debugging procedures, you might consider using an index card so that you can see only the line you are reading. Perhaps this technique will help you avoid the common error of assuming that the procedure does what you want it to do instead of what you told Logo to do.

When using an IF with three inputs to build a loop, be particularly careful that you don't accidentally introduce embedded recursion. Embedded recursion is a little harder to see when you are using a three-input IF. The recursive call must be the last instruction to run. Consider this example:

```
TO EXAMPLE
PRINT [THIS IS AN DEMONSTRATION]
IF (READCHARACTER = "Y) [EXAMPLE CLEARTEXT] [PRINT [BYE]]
END
```

This procedure has embedded recursion. Each time the user types "Y," EXAMPLE is run again, but the CLEARTEXT is not run. When the user types something other than "Y," the "PRINT [BYE]" is run. Although it would be very difficult to tell, the CLEARTEXT is then run as many times as the user types "Y." (Try replacing the CLEARTEXT with a PRINT instruction.) This problem can be corrected by changing the procedure as follows:

```
TO EXAMPLE
PRINT [THIS IS A DEMONSTRATION]
IF (READCHARACTER = "Y) [CLEARTEXT EXAMPLE] [PRINT [BYE]]
END
```

EXAMPLE is the last instruction run if the user types "Y," and the procedure is now tail recursive.

In the above example, note that it is easy to get confused about brackets when you have a PRINT instruction. In the last line of the procedure EXAMPLE you see

```
[PRINT [BYE]]
```

Note that the first and last bracket are there because of the IF instruction, and the brackets around the word "BYE" are there because they are needed for the PRINT command.

Project Suggestion

You have written programs using the two-input IF. Perhaps you wished for a three-input IF. Go back to one of those programs and modify it by using the three-input form of IF to make it more powerful or interesting.

26. Procedure Inputs

Have you ever made a design out of similar shapes that differed only in size? For example, have you made a design made of squares or triangles of different size? Or, perhaps, have you made a square house with square windows? To make squares of several sizes, then you had to write several square procedures that made the turtle go forward different amounts. Now you are going to learn how to use one procedure to make shapes of any size you wish.

To make shapes of different sizes, you use procedure inputs. Procedure inputs are names that stand for values that can be changed each time the procedure is used. For example, if you write

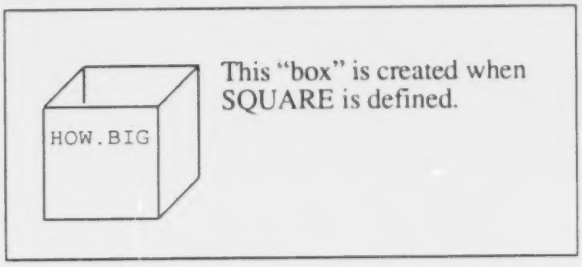
```
TO SQUARE :HOW.BIG
REPEAT 4 [FORWARD :HOW.BIG RIGHT 90]
END
```

and then type

```
SQUARE 50
```

the name HOW.BIG now has the value 50 associated with it. Then, when FORWARD needs an input, it finds :HOW.BIG. The ":" (called dots) in front of HOW.BIG tells Logo to use the value associated with HOW.BIG. Since 50 is associated with "HOW.BIG," the turtle moves forward 50 turtle steps.

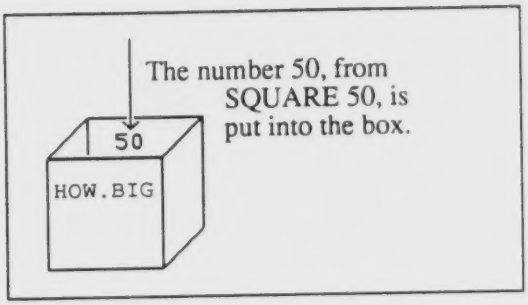
Another way to think about this is to imagine a box that is created when SQUARE is defined. The box is empty and has the word HOW.BIG written on the outside.



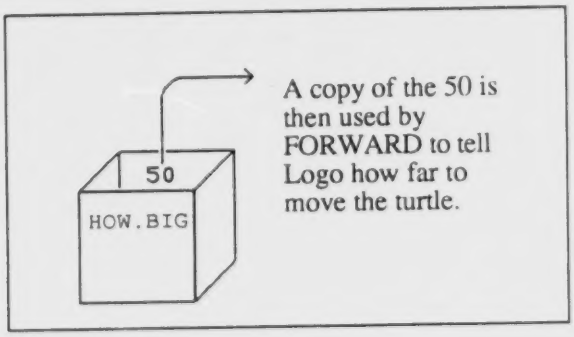
Then, when you type

```
SQUARE 50
```

the number 50 is put in the box.



Later, when FORWARD finds :HOW.BIG as input, Logo looks in the HOW.BIG box to get the value to give FORWARD as input.



You can now use the SQUARE procedure listed earlier to make designs. For example, if you type

```
TO DESIGN1
SQUARE 20
SQUARE 25
SQUARE 30
SQUARE 35
SQUARE 40
END
```

you see

or if you type

```
TO DESIGN2
SQUARE 30
RIGHT 10
SQUARE 35
RIGHT 10
SQUARE 40
RIGHT 10
```

```
SQUARE 45
END
```

you see

Whenever a procedure is written using a procedure input, you must always provide the value for the input name when using that procedure. For example, if you type

```
SQUARE
```

and press Enter, Logo will say

```
SQUARE needs more inputs
```

just as it does when you type FORWARD without an input.

Looping Using a Procedure Input

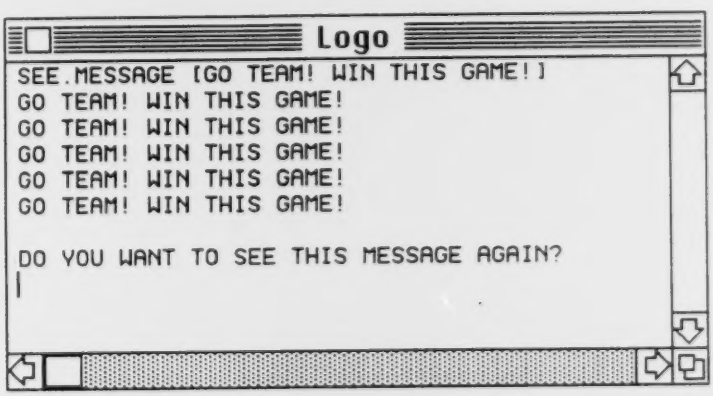
You have written procedures that ask the user if s/he wants to see a message again. Using procedure inputs, you can put any message you want in such a procedure, for example,

```
TO SEE.MESSAGE :MESSAGE
REPEAT 5 [PRINT :MESSAGE]
PRINT []
PRINT [DO YOU WANT TO SEE THE MESSAGE AGAIN?]
IF READCHARACTER = "Y [SEE.MESSAGE :MESSAGE]
END
```

To run this procedure, be sure you are in the Logo Window and type

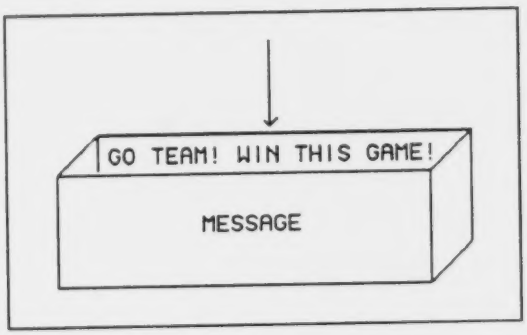
```
SEE.MESSAGE [GO TEAM!  WIN THIS GAME!]
```

You see


Logo
SEE.MESSAGE [GO TEAM! WIN THIS GAME!]
GO TEAM! WIN THIS GAME!
GO TEAM! WIN THIS GAME!
GO TEAM! WIN THIS GAME!
GO TEAM! WIN THIS GAME!
GO TEAM! WIN THIS GAME!
DO YOU WANT TO SEE THIS MESSAGE AGAIN?

If you type Y, then the same lines are repeated. Do you see what is happening?

- The words "GO TEAM! WIN THIS GAME!" are put in the "box" labeled "MESSAGE."



- Next, PRINT has as input :MESSAGE, read "dots MESSAGE." (The ":" symbol can be thought of as "look in the box.")
- PRINT finds "GO TEAM! WIN THIS GAME!" in the "MESSAGE box" and prints that value five times in the Logo Window.
- Next, READCHARACTER causes the procedure to wait for input. If "Y" is typed, then the "value in the MESSAGE box" is sent back to procedure SEE.MESSAGE, using the name MESSAGE.

You can make this procedure more user-friendly by using the three-input IF:

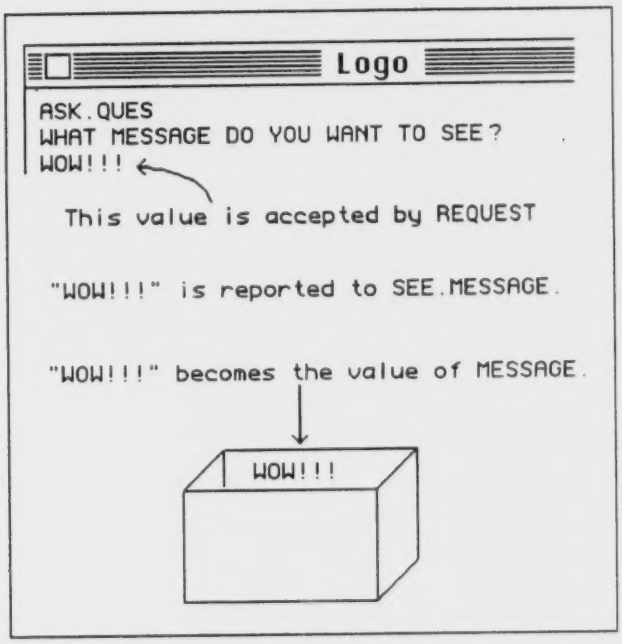
```
TO SEE.MESSAGE :MESSAGE
REPEAT 15 [PRINT :MESSAGE]
PRINT []
PRINT1 [DO YOU WANT TO SEE THIS MESSAGE AGAIN?]
IF READCHARACTER = "Y →
      [SEE.MESSAGE :MESSAGE]→
      [PRINT [O.K., GOODBYE!]]
END
```

Passing Values from the Keyboard to a Procedure Input

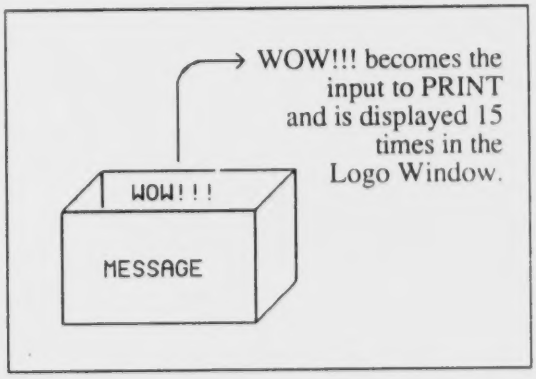
Adding one more procedure makes the program even more user-friendly:

```
TO ASK.QUES
PRINT [WHAT MESSAGE DO YOU WANT TO SEE?]
SEE.MESSAGE READLIST
END
```

Now, whatever is reported by READLIST is put in the MESSAGE box



and then used by the procedure SEE.MESSAGE



Take a few moments to be sure you understand what is happening in these procedures.

Creating a CASE Statement

Earlier you encountered the problem of creating a menu with two items using IF. If you run the procedure MENU below, all is well if you select either "1" or "2," but if you type any other number, the ASK.QUESTION procedure runs, even though that is probably not what you want to happen:

```
TO MENU
PRINT [1. SEE A PICTURE]
PRINT [2. ANSWER A QUESTION]
IF READWORD = 1 [PICTURE] [ASK.QUESTION]
END
```

Sometimes you want a procedure to allow multiple choices and, in addition, to do nothing at all if an incorrect choice is made. To accomplish this task, you must write a more elaborate program. Suppose you have the following menu procedure:

```
TO SHOW.CHOICES
PRINT [1. SEE A SQUARE]
PRINT [2. SEE A STAR]
PRINT [3. SEE A TRIANGLE]
END
```

You can then create a procedure to handle all the choices. This type of structure is called a CASE statement:

```
TO WHICH.ONE :CHOICE
IF :CHOICE = 1 [SQUARE STOP]
IF :CHOICE = 2 [STAR STOP]
IF :CHOICE = 3 [TRIANGLE STOP]
END
```

Now you can write

```
TO MENU
SHOW.CHOICES
WHICH.ONE READCHARACTER
END
```

Think about how this procedure works.

- First the menu is displayed on the Logo Window.
- Then the procedure WHICH.ONE is called.
- READCHARCTER waits for input from the keyboard and passes the value typed into the name CHOICE in procedure WHICH.ONE.
- CHOICE is then tested by each IF instruction. If there is a match, then the appropriate list of instructions is run.

The STOP primitive is not usually necessary in this case structure. However, if you have a lot of choices, it prevents the procedure from running though all possible matches listed in the procedure. STOP causes only the procedure in which it is located to stop running.

Tips and Techniques

There are a lot of new ideas in this chapter. If the idea of procedure inputs is new to you, then start with simple procedures like the ones for drawing polygons. Only when you understand these simple procedures should you try to write procedures with inputs that use interactive reporters and/or conditionals (IF).

The model used for procedure inputs in this chapter is boxes containing values. The value of this model is that it is easy to draw and explain. However, it has a serious flaw. A Logo name can be associated with only one value; that is, the "box" can contain only one value. However, boxes in the real world can contain more than one item. Further, when you pass a value to another name, it is still associated with the original name. A copy is passed. But when you take

something out of a real-world box, it is gone. Be careful that you don't take the model of values put into boxes too literally. Keep in mind that a Logo name can hold only one value and that it passes a copy of that value to another name.

Using PRINT instructions in procedures that use inputs can be very helpful in debugging. For example, you can use a PRINT instruction to verify the value of an input:

```
TO SQUARE :HOW.BIG
(PRINT [THE VALUE OF "HOW.BIG IS] :HOW.BIG)
REPEAT 4 [FORWARD :HOW.BIG RIGHT 90]
END
```

You can keep track of how often a procedure is run with a given input:

```
TO SEE.MESSAGE :MESSAGE
(PRINT [BEGINNING OF SEE.MESSAGE WITH VALUE] :MESSAGE)
CLEARTEXT
REPEAT 5 [PRINT :MESSAGE]
PRINT []
PRINT [DO YOU WANT TO SEE THE MESSAGE AGAIN?]
IF READCHARACTER = "Y [SEE.MESSAGE :MESSAGE]
END
```

or verify what value was passed to a procedure:

```
TO WHICH.ONE :CHOICE
(PRINT [THE VALUE PASSED TO CHOICE IS] :CHOICE)
IF :CHOICE = 1 [SQUARE STOP]
IF :CHOICE = 2 [STAR STOP]
IF :CHOICE = 3 [TRIANGLE STOP]
END
```

Using PRINT to trace the action of your procedures can save you a lot of debugging time. This is a case where you should let Logo do some of the debugging work for you.

Project Suggestions

Write procedures to draw a design made up of squares and triangles of all sizes. Be sure that you use an input and write only one SQUARE and one TRIANGLE procedure.

Write a program that asks the user what size design s/he wants to see. Then draw the design using the number the user types as input.

Think about a procedure to draw a triangle. You use REPEAT 3 followed by some commands. To draw a square, you use REPEAT 4. What would you use to draw a five-sided figure? Can you write a procedure to draw a figure with any number of sides using an input? What angle will you turn? (Hint: The turtle must turn 360 degrees to get back to where it started.)

27. More on Multiple Turtles

You have used multiple turtles. Recall that you can have several turtles active at once, all doing the same thing. You saw that

```
TELL number or [list of numbers]
```

tells which turtle or turtles should become active.

There are several other Logo primitives that are used with multiple turtles. Sometimes when you are working with more than one turtle, you forget which turtles are active. That problem is easily solved with WHO. Recall that WHO is a reporter that returns the numbers of the active turtles. Thus, if you type

```
TELL [1 3]
```

and then later in your work you forget which turtles are active, you can type

```
PRINT WHO
```

and Logo will display

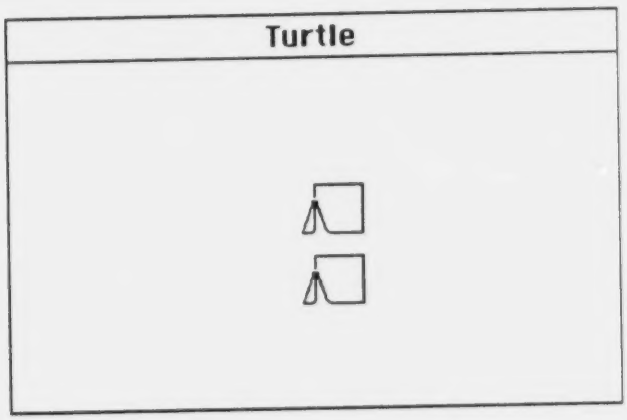
```
1 3
```

in the Logo Window.

Another useful command is EACH. EACH is followed by a list of commands for the active turtles to run. If you type

```
TELL 0
PENUP
BACK 30
PENDOWN
TELL [1 0]
SHOWTURTLE
SQUARE 20
```

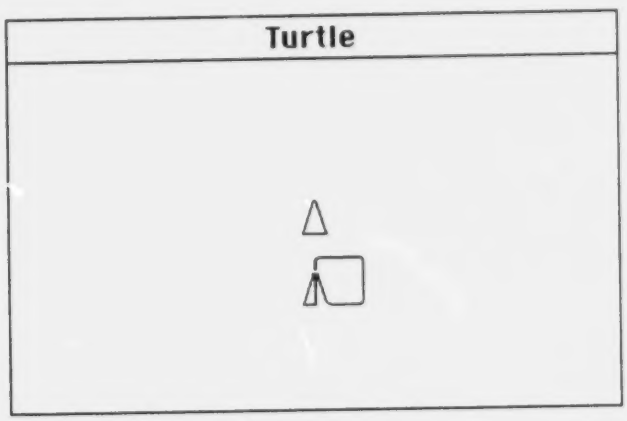
then turtles 1 and 0 draw a square *at the same time* (assuming you have defined a procedure SQUARE). You see



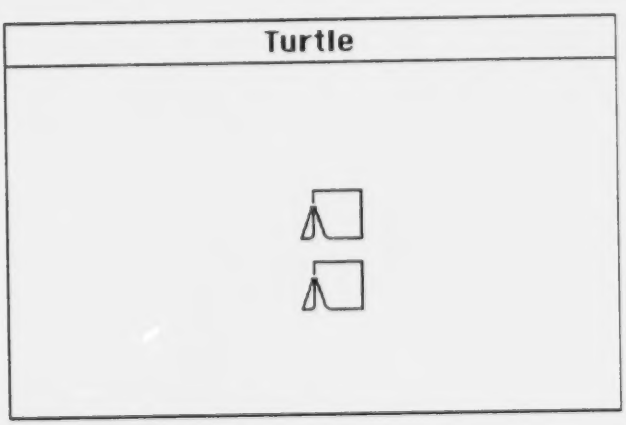
However, if you then type

```
EACH [SQUARE 20]
```

you see



followed by



Turtle 1 draws a square, and then turtle 0 draws a square. Notice the difference. If you issue commands after a TELL, all the active turtles move at once. If you use EACH followed by a list of commands, the turtles move one at a time. Also note that the turtles move in the *order* they appear in the WHO list.

A third useful command is ASK. ASK needs two inputs: a turtle number or list of turtle numbers, and a list of commands. It allows you to address either an active or an inactive turtle. Here is a list of commands to illustrate these ideas:

```
TELL [1 3]
SHOWTURTLE
```

If you type

```
PRINT WHO
```

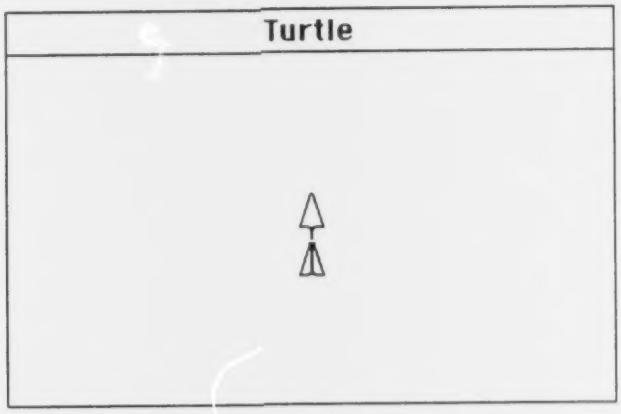
you see

```
1 3
```

If you type

```
ASK 2 [SHOWTURTLE BACK 20]
```

you see



but if you type

```
PRINT WHO
```

you still see

```
1 3
```

because ASK does not change the list of active turtles.

ASK communicates with turtle 2 and instructs it to move back 20 turtle steps. However, the list of active turtles is unchanged. Only TELL changes the list of active turtles reported by WHO.

Using these new commands and interactive commands, you can write an interactive program to make the turtles dance. First, you need a dance procedure:

```
TO DANCE
REPEAT 20 [FORWARD 1 LEFT 18 WAIT 2]
REPEAT 20 [BACK 1 RIGHT 18 WAIT 2]
REPEAT 10 [FORWARD 1 WAIT 4]
REPEAT 10 [BACK 1 WAIT 4]
REPEAT 20 [FORWARD 1 RIGHT 18 WAIT 2]
REPEAT 20 [BACK 1 LEFT 18 WAIT 2]
END
```

Next, you need to place the turtles in a row in the Turtle Window:

```
TO SET.UP
TELL [0 1 2 3]
CLEARGRAPHICS
PENUP
SHOWTURTLE
TELL 0
RIGHT 90
FORWARD 40
LEFT 90
TELL 2
LEFT 90
FORWARD 40
RIGHT 90
TELL 3
LEFT 90
FORWARD 80
RIGHT 90
END
```

Now you need to create the main procedure,

```
TO ASK.QUES
SET.UP
ASK.USER
END
```

and the ASK.USER procedure,

```
TO ASK.USER
PRINT []
PRINT [WHICH TURTLES DO YOU WANT TO DO A DANCE? (0, 1, 2, or 3)]
TELL READLIST
PRINT []
PRINT [ALL AT ONCE OR SEPARATELY?  (A=ALL AT ONCE S=SEPARATELY)]
IF READCHARACTER = "A [DANCE] [EACH [DANCE]]
PRINT []
PRINT [WHICH TURTLE WOULD YOU LIKE TO DANCE A SOLO?]
ASK READCHAR [DANCE]
PRINT []
PRINT [WOULD YOU LIKE TO SEE SOME MORE DANCING?]
IF READCHARACTER = "Y [ASK.USER]
END
```

Using a procedure input, you could even change the speed of the dance:

```
TO DANCE :SPEED
REPEAT 20 [FORWARD 1 LEFT 18 WAIT :SPEED]
REPEAT 20 [BACK 1 RIGHT 18 WAIT :SPEED]
REPEAT 10 [FORWARD 1 WAIT 2 + :SPEED]
REPEAT 10 [BK 1 WAIT 2 + :SPEED]
REPEAT 20 [FORWARD 1 RIGHT 18 WAIT :SPEED]
REPEAT 20 [BACK 1 LEFT 18 WAIT :SPEED]
END
```

Then, you could replace each DANCE in ASK.USER with ASK.DANCE:

```
TO ASK.DANCE
PRINT [HOW FAST SHOULD THE DANCE BE?]
DANCE READCHARACTER
PRINT []
END
```

What other modifications might you make to this program?

Tips and Techniques

The differences among TELL, EACH, and ASK can be confusing at first. Here is a brief summary to help you understand each command.

- TELL sets up the list of active turtles.
- EACH causes the list of active turtles to perform some action.
- ASK ignores the list of active turtles and directs any turtles to perform some action.

Why do you need each of these instructions?

- TELL is used to change the list of active turtles. It has no other function. Turtle commands given *after* a TELL command cause all the turtles to do those commands *at the same time*.
- EACH is used if you want the turtles to perform some action *one at a time*.
- ASK is used to give direction to turtles *not in the active list*. For example, you might "ASK" a non-active turtle to return its color, shape, or heading.

The ASK.USER procedure in this chapter combines a lot of ideas you have learned in previous chapters. It is also a good example of the kind of procedure where inserting PRINT instructions can help in debugging. Here is the ASK.USER procedure with a number of possible diagnostic instructions inserted that might help in the debugging:

```
TO ASK.USER
PRINT [AT THE BEGINNING OF ASK.USER]
PRINT []
PRINT [WHICH TURTLES DO YOU WANT TO DO A DANCE (0, 1, 2, 3)?]
TELL READLIST
PRINT WHO   <-This shows which turtles are now active.
PRINT []
PRINT [ALL AT ONCE OR SEPARATELY?  (A=ALL AT ONCE S=SEPARATELY)]
IF READCHARACTER = "A →
      [PRINT [A SELECTED] DANCE] →
      [PRINT [A NOT SELECTED] EACH [DANCE]]
PRINT []
PRINT [WHICH TURTLE WOULD YOU LIKE TO DANCE A SOLO?]
ASK READCHAR [(PRINT [TURTLE PICKED IS] WHO) DANCE] <-WHO used inside the ASK returns
                                                      the turtle active at that moment.
PRINT []
PRINT [WOULD YOU LIKE TO SEE SOME MORE DANCING?]
IF READCHARACTER = "Y [ASK.USER]
END
```

Take a few moments to examine where the diagnostic instructions have been added. Learning to put such instructions in useful places takes practice and is part of the art of debugging.

Project Suggestion

Write an interactive program using multiple turtles and inputs. Perhaps you could have the turtle draw geometric shapes or draw different colored designs. Start with a simple idea and expand it as you get more ideas.

28. Using Inputs to Pass Values Among Procedures

You now know how to write procedures with inputs. You can write one procedure to draw squares of any size or print a variety of messages. You also know how to send a value typed by a user to a procedure with an input. However, what if you want to send a value typed by the user to several procedures? For example, suppose you have a procedure that asks the user to tell how big s/he wants some shapes. You can't write

```
TO ASK.QUES
PRINT [WHAT SIZE SHAPES?]
SQUARE READWORD <—Remember, you have to define or load READWORD.
TRIANGLE READWORD
END
```

Do you see why? If you type

```
ASK.QUES
```

Logo puts

```
WHAT SIZE SHAPES?
```

in the Turtle Window. If the user then types

```
50
```

then a square of size 50 is drawn. Everything seems fine so far.

Next, the procedure stops and waits for a second input because it encounters another READWORD. But you wanted a square and triangle of the same size. The user should not have to type 50 again.

Here is one way to solve this problem:

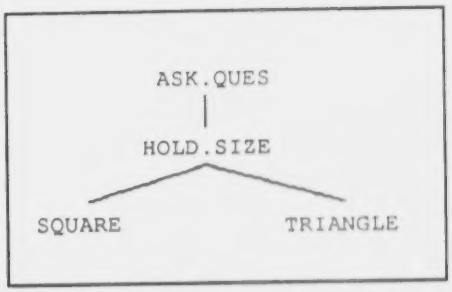
```
TO ASK.QUES
PRINT [WHAT SIZE SHAPES?]
HOLD.SIZE READWORD
END

TO HOLD.SIZE :SIZE
SQUARE :SIZE
TRIANGLE :SIZE
END

TO SQUARE :HOW.BIG
REPEAT 4 [FORWARD :HOW.BIG RIGHT 90]
END

TO TRIANGLE :LENGTH
REPEAT 3 [FORWARD :LENGTH RIGHT 120]
END
```

Do you see how HOLD.SIZE works? Examine the procedure tree to help you see the structure of the program. Notice that procedure inputs do not appear on the procedure tree.



Carefully study the following explanation of how ASK.QUES works. Take special note that the same input name is not used in any two procedures. While it is legal to use the same name, keeping the names different helps you to think about what is happening to the values of the names as the program runs. It also helps you avoid future bugs in more complex programs.

If you type

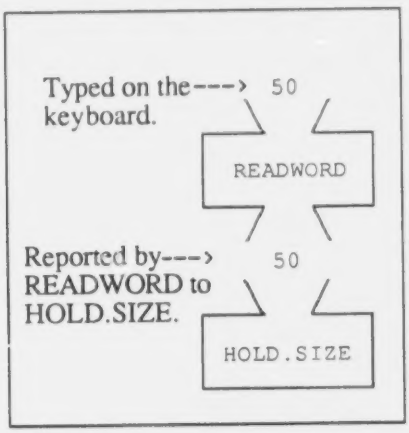
```
ASK.QUES
```

Logo responds with

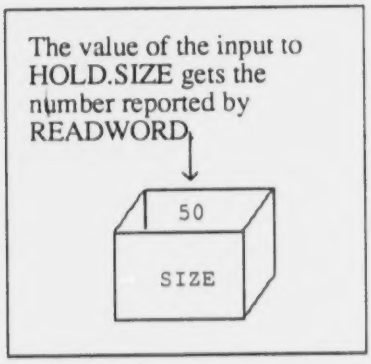
```
WHAT SIZE SHAPES?
```

Suppose you type

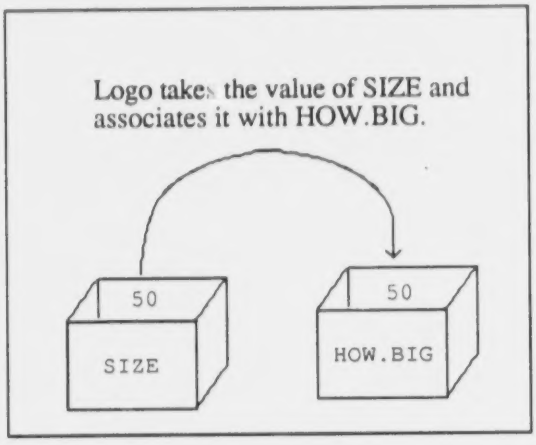
```
50
```



Then READWORD reports 50 to HOLD.SIZE. HOLD.SIZE needs an input. Fifty becomes the input to the HOLD.SIZE procedure. Thus, SIZE has the value of 50 in HOLD.SIZE.



Next, SQUARE is called in the procedure HOLD.SIZE. SQUARE needs an input. Recall that ":SIZE" is read "the value in the box SIZE." Logo then passes the value associated with SIZE into the HOW.BIG box.

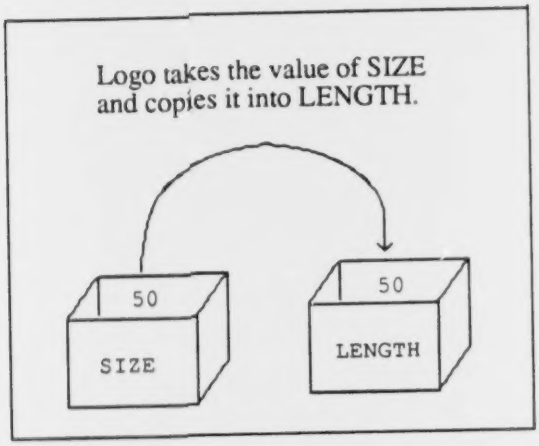


In the procedure SQUARE, the instruction

```
FORWARD :HOW.BIG
```

instructs Logo to look for the "value in the box HOW.BIG" to see how far forward to move the turtle.

Finally, the procedure TRIANGLE is called. In a similar manner, Logo copies the value of HOW.BIG into the LENGTH box.



Now the value of LENGTH is used to move the turtle forward to draw a triangle.

The above explanation details a rather complex process computer scientists call parameter passing. If you didn't understand it completely, don't be concerned. As you work with this idea, it will be become clearer simply by modeling it and practicing it. You should read the explanation several times, and return to it in the future when you encounter similar examples. When you can draw the diagrams yourself and explain what is happening, you will have made this idea your own. You will then be able to use it in a variety of situations.

Here is another example. Can you explain what is happening at each step of the process?

```
TO GREET
PRINT [WHAT IS YOUR NAME?]
HOLD.NAME READLIST
END

TO HOLD.NAME :NAME
PRINT.GREETING :NAME
SHOW.DESIGN
SAY.FAREWELL :NAME
END

TO PRINT.GREETING :WHO
(PRINT [HI,] :WHO)
PRINT [HERE IS A NEAT DESIGN.]
END

TO SHOW.DESIGN
 (some graphics here)
END

TO SAY.FAREWELL :TO.WHOM
(PRINT [SO LONG, ] :TO.WHOM)
PRINT [HOPE YOU ENJOYED THE GRAPHICS.]
END
```

Did you conclude that READLIST passes a value to NAME in HOLD.NAME? Then the value of NAME is copied into WHO in PRINT.GREETING. Finally, the value of NAME is copied into TO.WHOM to be used in the procedure SAY.FAREWELL.

Notice that the name of the user is used in the first and last procedure in HOLD.NAME, but not in the design procedure. This technique allows you to use a piece of information anywhere you want in a program.

Tips and Techniques

The caution about the "box" model in Chapter 26 applies here as well. A Logo name can hold only one value at a time. When a value is passed to another name, a copy is passed; the original name keeps the value as well.

When you begin to use procedure inputs in a number of procedures in your program, it is important that you check each procedure carefully to be sure it runs by itself. For example, you can check the HOLD.NAME procedure

```
TO HOLD.NAME :NAME
PRINT.GREETING :NAME
SHOW.DESIGN
SAY.FAREWELL :NAME
END
```

by typing something like

```
HOLD.NAME "MARY
```

You can check the PRINT.GREETING procedure

```
TO PRINT.GREETING :WHO
(PRINT [HI,] :WHO)
PRINT [HERE IS A NEAT DESIGN.]
END
```

by typing

```
PRINT.GREETING "MARY
```

If each procedure doesn't run by itself, then you are probably using procedure inputs incorrectly.

Project Suggestions

Draw the diagrams and "boxes" to explain the program GREET.

Practice writing programs that use a value typed at the keyboard. At first, use the models above, but then see if you can develop a program that goes beyond the models given in this chapter.

29. Incrementing

Have you ever wanted to write a program that counted something? Perhaps you wanted the user to say how many times s/he wanted a message to appear. Or perhaps you wanted to write a program to see all the shapes you have in memory. You can write programs that count, using recursion. Begin with this procedure:

```
TO COUNT.UP :INPUT
PRINT :INPUT
WAIT 5
COUNT.UP :INPUT + 1
END
```

If you type

```
COUNT.UP 1
```

Begin with a value of 1 for INPUT, and substitute values:

```
TO COUNT.UP 1
PRINT 1
WAIT 5
COUNT.UP 1 + 1 → TO COUNT.UP 2
END                PRINT 2
                   WAIT 5
                   COUNT.UP 2 + 1 → TO COUNT.UP 3
                   END                PRINT 3
                                      WAIT 5
                                      COUNT.UP 3 + 1 → and so forth
                                      END
```

Logo responds with

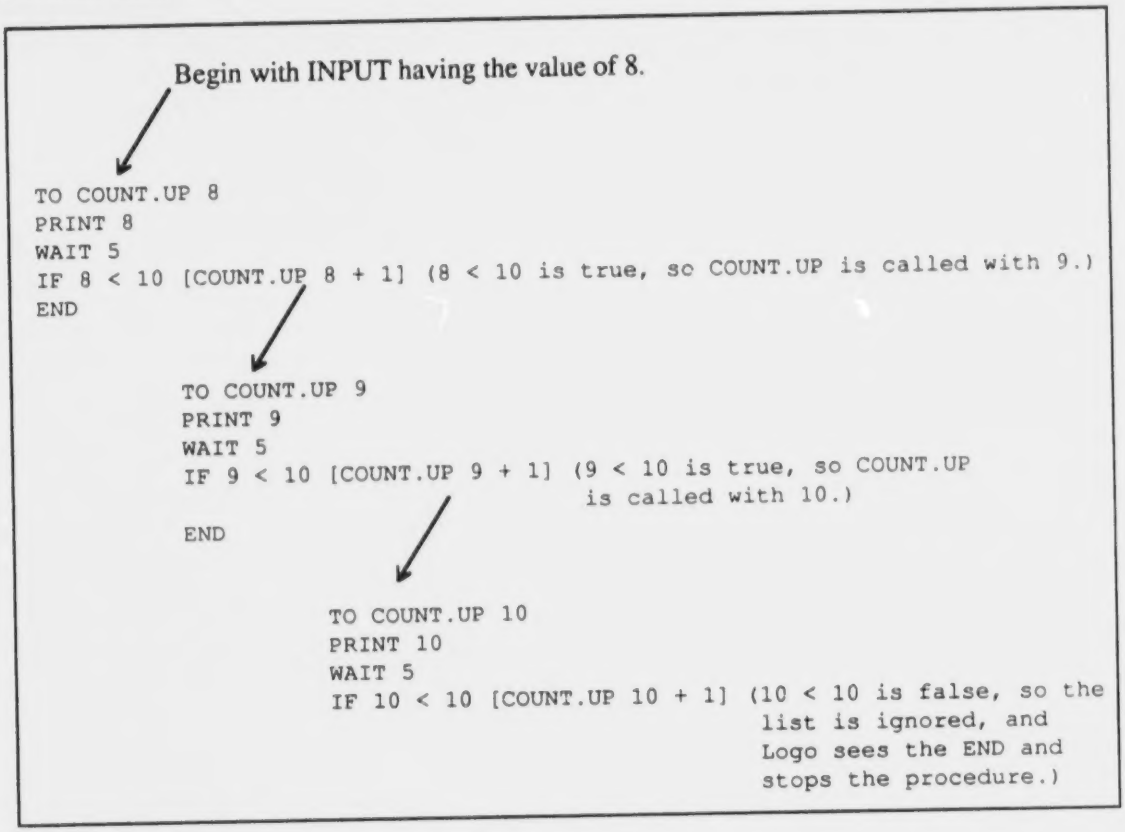
```
1
2
3
4
5
.
.
.
```

forever! There is no stop rule. You must press Command-G to stop the procedure. Before thinking about how to stop this procedure, examine carefully what is happening.

To stop the procedure at 10, you can add an IF instruction:

```
TO COUNT.UP :INPUT
PRINT :INPUT
WAIT 5
IF :INPUT < 10 [COUNT.UP :INPUT + 1]
END
```

Now the procedure will add 1 to the input value until it reaches 10. Examine the last several procedure calls.

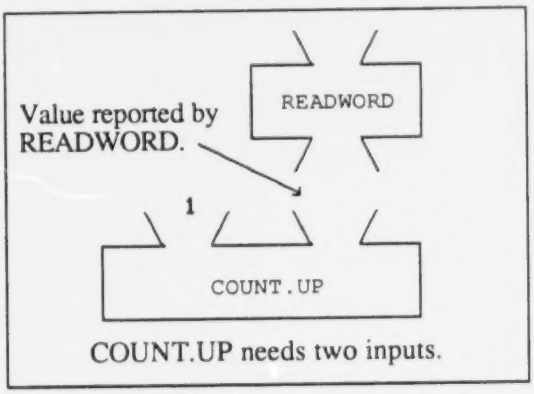

Begin with INPUT having the value of 8.
TO COUNT.UP 8
PRINT 8
WAIT 5
IF 8 < 10 [COUNT.UP 8 + 1] (8 < 10 is true, so COUNT.UP is called with 9.)
END
TO COUNT.UP 9
PRINT 9
WAIT 5
IF 9 < 10 [COUNT.UP 9 + 1] (9 < 10 is true, so COUNT.UP is called with 10.)
END
TO COUNT.UP 10
PRINT 10
WAIT 5
IF 10 < 10 [COUNT.UP 10 + 1] (10 < 10 is false, so the list is ignored, and Logo sees the END and stops the procedure.)

This procedure can now be used in a program that asks the user how high s/he wants to count:

```
TO ASK.COUNT
PRINT [HOW HIGH DO YOU WANT TO COUNT?]
COUNT.UP 1 READWORD
END

TO COUNT.UP :INPUT :MAX
PRINT :INPUT
WAIT 5
IF :INPUT < :MAX [COUNT.UP (:INPUT + 1) :MAX]
END
```

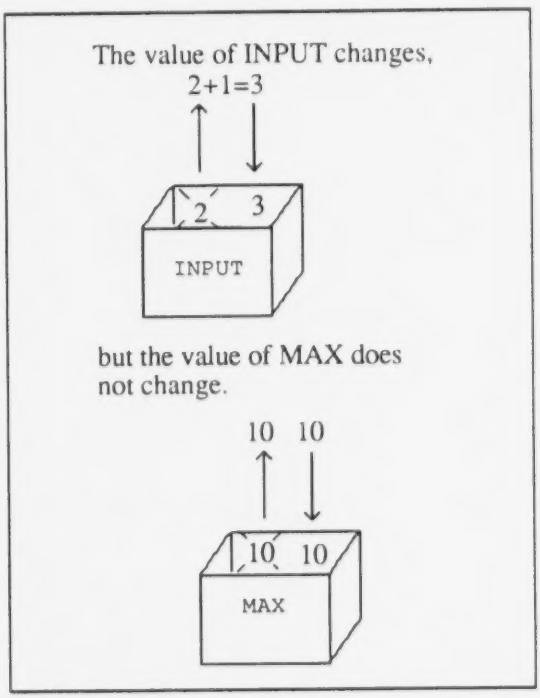
Notice that the procedure now has two inputs: the number printed in the procedure and the maximum.



COUNT.UP needs two inputs.

When COUNT.UP is called, the number 1 is put into the "INPUT box." The number typed by the user (and reported by READWORD) is put in the "MAX box." At the end of the procedure, the IF instruction checks to see if the number input is less than the maximum. If so, then the procedure is called recursively.

Look carefully at this recursive call. The first value sent back to COUNT.UP is :INPUT + 1. That is, one is added to the initial input so that when the procedure runs again, it runs with a number one larger than the original input. The value of MAX is passed unchanged.



Note that in the stop rule, (:INPUT + 1) is in parentheses. The parentheses are not necessary. They are only there to make the procedure more readable.

The programs shown so far count by 1s–that is, 1 is added each time the procedure is called. What if you wanted to count by 5s or 10s? Instead of adding 1 each time, you would add 5 or 10. COUNT.UP could be modified as follows:

```
TO COUNT.UP :INPUT :TO.ADD :MAX
PRINT :INPUT
WAIT 5
IF :INPUT < :MAX
      [COUNT.UP (:INPUT + :TO.ADD) :TO.ADD :MAX]
END
```

You can take this one step further by allowing the user to input the amount to add:

```
TO ASK.COUNT
PRINT [TYPE THE AMOUNT YOU WANT TO]
PRINT [ADD FOLLOWED BY ENTER.]
PRINT [THEN TYPE THE HIGHEST NUMBER YOU]
PRINT [WANT THE PROGRAM TO PRINT, AND]
PRINT [PRESS ENTER.]
COUNT.UP 7 READWORD READWORD
END
```

Think carefully about what is happening. Now there are three inputs about which you need to be concerned. If you type

```
ASK.COUNT
```

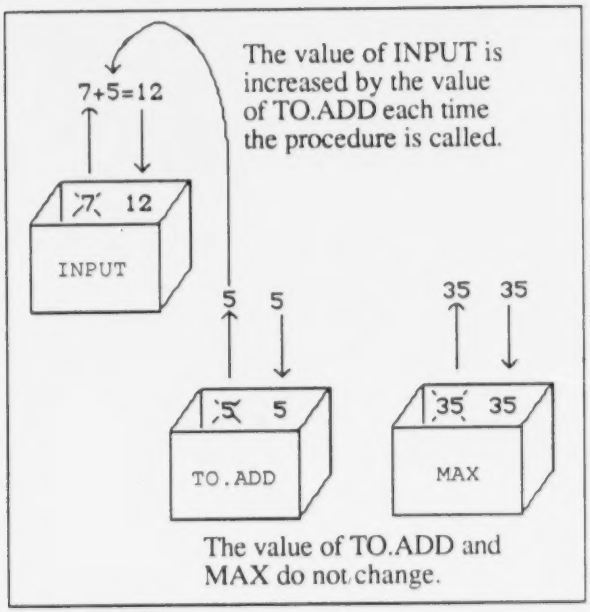
Logo responds with

```
TYPE THE AMOUNT YOU WANT TO
ADD FOLLOWED BY ENTER.
THEN TYPE THE HIGHEST NUMBER YOU
WANT THE PROGRAM TO PRINT, AND
PRESS ENTER.
```

If you then type

```
5
35
```

COUNT.UP is called with a value of 7 for INPUT, 5 for TO.ADD, and 35 for MAX.



Can you predict what the output from this program will look like? There is a bug in this program. It does not perform exactly as described. Spend some time thinking through each recursive call to the procedure. Try different input values. Can you find the bug?

One further modification of the procedure allows the user to input the number to start with, namely, the value of INPUT:

```
TO ASK.COUNT
PRINT [TYPE THE AMOUNT YOU WANT TO]
PRINT [ADD FOLLOWED BY ENTER.]
PRINT [THEN TYPE THE HIGHEST NUMBER YOU]
PRINT [WANT THE PROGRAM TO PRINT, AND]
PRINT [PRESS ENTER.]
PRINT [THEN TYPE THE HIGHEST NUMBER YOU]
PRINT [WANT THE PROGRAM TO PRINT, AND]
PRINT [PRESS RETURN.]
COUNT.UP READWORD READWORD READWORD
END

TO COUNT.UP :INPUT :TO.ADD :MAX
PRINT :INPUT
WAIT 5
IF :INPUT < :MAX [COUNT.UP (:INPUT + :TO.ADD) :TO.ADD :MAX]
END
```

Can you draw the diagrams to explain this procedure?

Notice that the value reported by the first READWORD is put in the "INPUT box," the value reported by the second READWORD is put in the "TO.ADD box," and the value reported by the third READWORD is put in the "MAX box." Both the MAX and the TO.ADD boxes remain unchanged as the procedure runs. As before, only the INPUT box changes as the amount to add each time is added.

Now the program is even more flexible. Try running it. This version has the same bug. It doesn't count as high as it says it does. Can you fix this bug?

Using Incrementing

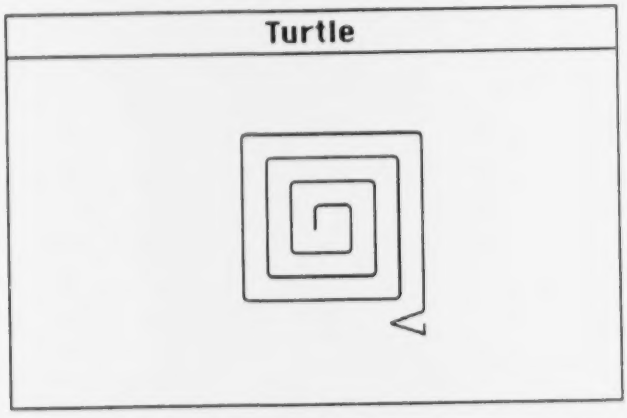
Incrementing can be used in a variety of settings. Suppose you wanted to make a "square spiral," that is, a square that gets longer as each side is drawn. You need to add a fixed amount each time you draw a side. This is similar to the COUNT.UP program above. You could write

```
TO GROW.SQUARE :SIDE
FORWARD :SIDE
RIGHT 90
IF :SIDE < 80 [GROW.SQUARE :SIDE + 5]
END
```

If you type

```
GROW.SQUARE 10
```

the turtle draws a side of length 10, then a side of length 15, then a side of length 20, and so forth until the side reaches 80.



How about writing a program to display all of the pen patterns? Recall that

```
SETPPATTERN 10
```

sets the pattern number 10. Sometimes it would be helpful to be able to see all the different pen patterns without changing the pen pattern a lot of times. You can write a procedure that steps through the patterns one at a time. You can start with pen pattern number 1 and add 1 until you reach the largest number, 38. This is similar to the COUNT.UP procedure:

```
TO SEE.PATTERNS :WHICH.ONE
CLEARGRAPHICS
SETPPATTERN :WHICH ONE
FORWARD 20
WAIT 60
IF :WHICH.ONE < 39 [SEE.PATTERNS :WHICH.ONE + 1]
END
```

When you type

```
SETPENSIZE 20 1
SEE.PATTERNS 1
```

you briefly see each pattern from the set of available patterns.

You can also use incrementing to count how often something happens, for example,

```
TO DRAW.STARS :COUNTER :HOW.MANY
PLACE.STAR <— a procedure to locate a star in the Turtle Window
STAR <— a procedure to draw a star
IF :COUNTER < :HOW.MANY→
      [DRAW.STARS (:COUNTER + 1) :HOW.MANY]→
      [(PRINT [YOU SEE] :HOW.MANY [STARS!])]
END
```

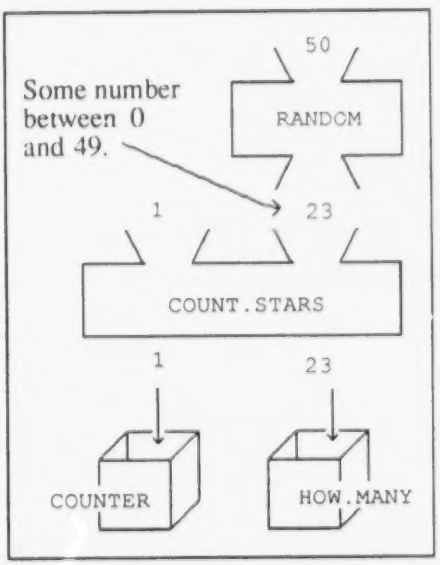
If you type

```
DRAW.STARS 1 20
```

this procedure prints 20 stars and then prints on the text screen that there are indeed 20 stars. This seems a bit strange. You could use this procedure in a different way. By giving the input HOW.MANY a random number as input, each time the procedure is run, a different number of stars is drawn. Then, printing the number drawn doesn't seem so unnecessary:

```
DRAW.STARS 1 (RANDOM 50)
```

In this procedure, RANDOM 50 provides the input for HOW.MANY, and COUNTER keeps track of the number of stars drawn.



Incrementing is a powerful tool. You can find many uses for it in your programs. Whenever you want to keep track of the number of times something occurs or want to repeatedly add the same number, you need to use the techniques explained in this section.

Embedded Recursion

When you first learned to construct loops in Logo using recursive calls, you were cautioned to be sure that the recursive call was the last instruction in your program or procedure. In chapter 19, you saw an example in which the last lines of a procedure were never run because the procedure simply called itself again and again. In this context, embedded recursion looks like it has no particular value at all. It is, however, a powerful tool used extensively in more advanced Logo programming.

"Counting" procedures provide a good opportunity to further examine embedded recursion. Consider a modification of the simplest COUNT.UP procedure from the beginning of this chapter, renamed now as DO.COUNT:

```
TO DO.COUNT :INPUT.VALUE
PRINT :INPUT.VALUE
DO.COUNT :INPUT.VALUE + 1
END
```

Next, modify this procedure so that it includes a stop rule:

```
TO DO.COUNT :INPUT.VALUE
PRINT :INPUT.VALUE
IF :INPUT.VALUE < 5 [DO.COUNT :INPUT.VALUE + 1]
END
```

This procedure will print the values from 1 to 5 if you enter

```
DO.COUNT 1
```

Next, make a simple change as follows:

```
TO DO.COUNT :INPUT.VALUE
IF :INPUT.VALUE < 5 [DO.COUNT :INPUT.VALUE + 1]
PRINT :INPUT.VALUE
END
```

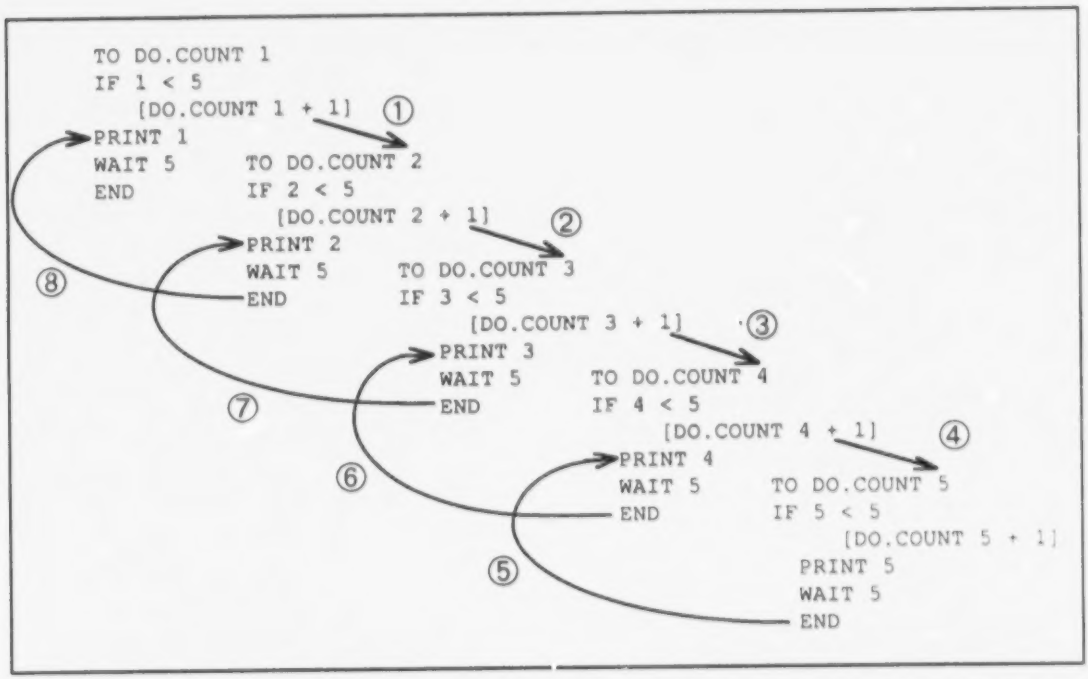
Now if you type

```
DO.COUNT 1
```

you see

```
5
4
3
2
1
```

What is happening? Examine the diagram below, which is like the one we looked at for COUNT.UP earlier.



First, procedure DO.COUNT is called. The number 1 is clearly less than the number 5. Thus, at step 1 in the diagram, DO.COUNT is called with an input of 2. Notice that the PRINT 1 instruction is not run at this time. Next, 2 is tested against 5. The test fails, and DO.COUNT is called with an input value of 3 (at step 2 in the diagram). This process continues, until, at step 4, 5 < 5 is false. Thus, DO.COUNT is not called again, so the procedure moves to the next instruction,

```
PRINT 5
```

Thus, 5 is printed. Logo reaches the END of the procedure. Now, the procedure above is incomplete (step 5). Logo finishes that procedure by running

```
PRINT 4
```

Now there remains the rest of the PRINT instructions that were not run, so Logo runs those instructions as it encounters them until it reaches an END primitive. Then it moves to the next unfinished procedure that was stacked up waiting to be run.

At first, embedded recursion seems quite mysterious. If you study the above example carefully, you will no doubt see why and how it works. However, embedded recursion is probably not something you want to use just now in your programming. If you are working with other people learning Logo, you will often see the results of their unwittingly using embedded recursion. Embedded recursion is often the reason a message flashes in the Logo Window again and again when the author thought it should appear only once. Perhaps the screen flashes at the end of a program for unexplained reasons, or there are unexpected pauses when a program should just stop. Such effects occur when commands such as CLEARTEXT or CLEARGRAPHICS are placed after a recursive call.

other people learning Logo, you will often see the results of their unwittingly using embedded recursion. Embedded recursion is often the reason a message flashes in the Logo Window again and again when the author thought it should appear only once. Perhaps the screen flashes at the end of a program for unexplained reasons, or there are unexpected pauses when a program should just stop. Such effects occur when commands such as CLEARTEXT or CLEARGRAPHICS are placed after a recursive call.

When you have had more experience, you may want to explore this idea further. For a more complete treatment of recursion, see Brian Harvey's book *Computer Science Logo Style, Volume I*, published by MIT Press.

Tips and Techniques

The procedures in the first part of this chapter become increasingly complex. If you are new to programming, you may well have trouble following the explanations. You can use the ideas you have learned about debugging to help you understand what is happening in a difficult procedure. For example, here is the last version of COUNT.UP with a number of PRINT instructions added to help you understand the procedure:

```
TO COUNT.UP :INPUT.VALUE :TO.ADD :MAX
PRINT :INPUT.VALUE
WAIT 10
(PRINT [MAX = ] :MAX)
(PRINT [TO.ADD = ] :TO.ADD)
IF :INPUT.VALUE < :MAX→
      [COUNT.UP (:INPUT.VALUE + :TO.ADD) :TO.ADD :MAX]
END
```

You might then run this procedure by typing

```
COUNT.UP 5 3 15
```

and studying the results. Then try changing each of the values to see what happens. Use this technique with any procedures you have trouble understanding.

Project Suggestions

The last example of COUNT.UP given in this chapter does not actually print the maximum value, for example, COUNT.UP 3 5 9. See if you can fix it so the procedure runs correctly for any values you input.

Draw appropriate diagrams for the various versions of COUNT.UP. Can you explain the behavior of each of the modifications to COUNT.UP?

Experiment with other types of GROW procedures. Make triangular or hexagonal spirals. What happens if the spirals wrap? Can you modify the procedure so that the user can tell how big the spiral should get? How about modifying it so the user can tell how large the space between sides of the spiral should be?

Modify the SEE.PATTERNS procedure so that the number of the pattern is printed under the block the turtle draws.

Modify the SEE.PATTERN procedure so that the user can repeatedly specify which pattern s/he wants to see. Hint: Add a question asking the user if s/he wants to see another pattern.

30. Logo Arithmetic and Math

Have you discovered that you can do math in Logo? For example, you can type

```
PRINT 3 + 4
```

and Logo prints

```
7
```

or

```
PRINT (3 * 4) - 10
```

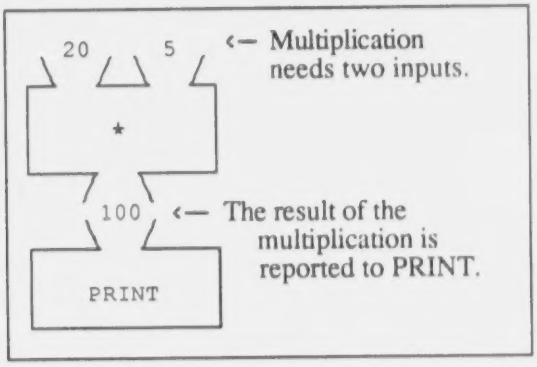
and Logo prints 2.

Notice that the asterisk (*) is the sign Logo uses for multiplication. Here are examples of the arithmetic and mathematics Logo can do:

PRINT 20 + 5---------- Logo prints 25
PRINT 20 - 5---------- Logo prints 15
PRINT 20 * 5---------- Logo prints 100
PRINT 20 / 5---------- Logo prints 4

Logo also includes a wide variety of more sophisticated mathematical functions. See the end of this chapter for a complete list.

Notice that all of these mathematical operations are reporters–they report a value to the PRINT command. For example, you can diagram PRINT 20 * 5 as follows:

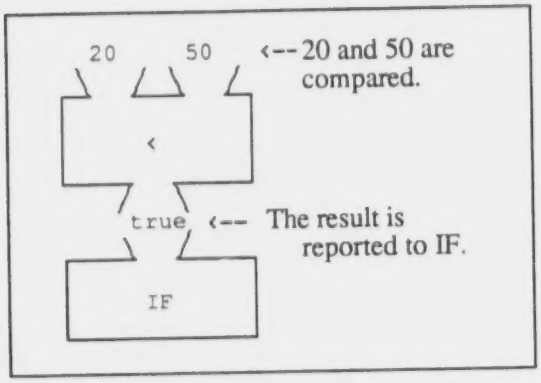


In math, these are called binary operations because they have two inputs.

You have used other arithmetic reporters with IF. The result of =, <, or > was reported to IF.

```
IF 20 < 30
```

would be diagrammed as:



All three of these reporters (<, >, =) return either TRUE or FALSE. When these are used in IF, the "true" or "false" is reported to IF. IF then selects an action based on the result reported (true or false).

You can print the result of these reporters just as you can print the result of arithmetic reporters:

PRINT 20 = 5	Logo prints "FALSE"
PRINT 20 < 5	Logo prints "FALSE"
PRINT 20 > 5	Logo prints "TRUE"

Using arithmetic reporters and what you know about incrementing, you can easily have Logo generate a table of values, for example,

```
TO SQUARES :NUMBER :LARGEST
(PRINT :NUMBER :NUMBER * :NUMBER)
IF :NUMBER < :LARGEST [SQUARES :NUMBER + 1 :LARGEST]
END
```

If you type

```
SQUARES 1 5
```

Logo prints

```
1 1
2 4
3 9
4 16
5 25
```

You can create multiplication tables or other tables easily.

With the skills you now have in Logo, you can write a simple arithmetic drill-and-practice program. Begin with a procedure to print a problem and accept the answer:

```
TO SEE.AND.CHECK :NUMBER1 :NUMBER2
PRINT []
PRINT []
(PRINT :NUMBER1 [+] :NUMBER2 [=])
IF READWORD = (:NUMBER1 + :NUMBER2)→
      [GOOD.JOB]→
      [TRY.AGAIN :NUMBER1 :NUMBER2]
END
```

Examine carefully the line that prints the problem. Do you see why it prints the problem and not the answer? How is

```
:NUMBER1 [+] :NUMBER2
```

different from

```
:NUMBER1 + :NUMBER2
```

Do you see that in the first line the plus sign is quoted (see Chapter 17)? In the second it is not. That means that the [+] will be printed. When + appears without being quoted, it is treated as a reporter. It reports the sum of the two numbers. This is an important idea. Study the above example to be sure you see what is happening.

The procedures GOOD.JOB and TRY.AGAIN might look like this:

```
TO GOOD.JOB
CLEARTEXT
REPEAT 50 [PRINT1 [GOOD FOR YOU!!!!!! ]]
END

TO TRY.AGAIN :N1 :N2
PRINT []
PRINT [NO, TRY THE PROBLEMS AGAIN.]
SEE.AND.CHECK :N1 :N2
END
```

Notice that the numbers were sent to TRY.AGAIN but were not used. They could then be sent back to SEE.AND.CHECK so that the same problem is printed again. The procedures SEE.AND.CHECK and TRY.AGAIN make use of the idea, which you have seen earlier, of passing values from one procedure to another.

To get SEE.AND.CHECK to select random problems, you need an instruction something like

```
SEE.AND.CHECK RANDOM 10 RANDOM 10
```

Each number in the problem shown then ranges from 0 to 9. Then, a procedure is needed to allow the user to select how many problems s/he wants to practice:

```
TO ASK.USER
PRINT [HOW MANY PROBLEMS DO YOU WANT TO DO?]
REPEAT READWORD [SEE.AND.CHECK RANDOM 10 RANDOM 10]
END
```

Now this program can be used to do arithmetic practice.

Tips and Techniques

You have seen that parentheses are often used in an IF instruction when the reporters <, >, or = are used. If you do not use parentheses, sometimes mysterious bugs appear. This is because the standard mathematical symbols are "infix" notation. The symbol goes in between the inputs; that is, we write 3 + 4 or 5 < 6 rather than +3 4 or < 5 6. All Logo commands that are reporters use "prefix" notation; that is, the inputs to the procedures or primitives come after the command or reporter. When infix and prefix notations are mixed, it can be difficult for the Logo parser to translate correctly. Parentheses force Logo to translate expressions in the way you intend.

Below is a list of the mathematical reporters available in *Terrapin Logo*. Notice that there are both prefix forms of the usual arithmetic operations and more sophisticated mathematical functions available.

`ABS` *number*	Returns the absolute value of its input
`ARCTAN` *number1 number2*	Returns the arctangent of the quotient of the two numbers given
`COS` *number*	Returns the cosine of an angle measured in degrees
`DIVIDE` *number1 number2*	Returns the quotient of the two numbers
`EXP` *number*	Returns *e* raised to the *number* power
`FLOOR` *number*	Returns the nearest integer not greater than *number*
`INTEGER`	Returns the integer part of *number*
`LN` *number*	Returns the natural logarithm
`MINUS` *number1 number2*	Returns the difference between two numbers
`PLUS` *number1 number2*	Returns the sum of two numbers
`POWER` *number1 number2*	Returns the *number1* raised to the *number2* power
`QUOTIENT` *number1 number2*	Returns the quotient of the two numbers
`REMAINDER` *number1 number2*	Returns the integer remainder of *number1* divided by *number 2*
`ROUND` *number*	Returns the nearest integer
`SIN` *number*	Returns the sine of *number*
`SQRT` *number*	Returns the positive square root of the input
`TIMES` *number1 number2*	Returns the product of *number1* and *number2*

Project Suggestions

Write a program to generate a table of values that you might find useful, such as a table of squares. Print the results to use elsewhere.

Try explaining the program SEE.AND.CHECK to someone who has never seen this idea before.

Modify the SEE.AND.CHECK program so that the user can give the range of numbers s/he wants.

31. OUTPUT

When you wrote the procedure READWORD to read numbers from the keyboard, you used the OUTPUT command even though you had not yet studied it. All of the procedures that *you* have written have been commands. That is, when you typed the name of the procedure, it caused some effect–it did not report a value. OUTPUT is the command that allows you to write your own reporters. Procedures containing OUTPUT report values. Think about READWORD. If you type

```
READWORD
```

and then type

```
HELLO
```

Logo responds

```
There is no procedure named HELLO
```

READWORD reported a value, but you did not tell Logo what you wanted to do with the value. READWORD behaves just like other reporters you have used, like RANDOM.

Suppose you wanted a procedure that reported the cube of a number. Remember that the cube of a number is that number times itself three times, for example,

$$2^3 = 2 \times 2 \times 2 = 8$$

and

$$5^3 = 5 \times 5 \times 5 = 125$$

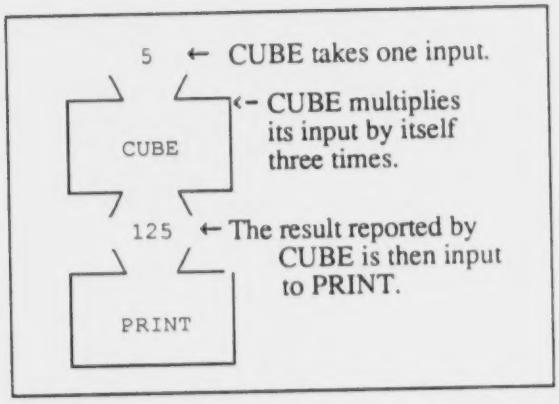
You can use OUTPUT to write such a procedure:

```
TO CUBE :NUMBER
OUTPUT :NUMBER * :NUMBER * :NUMBER
END
```

Now, if you type

```
PRINT CUBE 5
```

Logo prints 125.


5
← CUBE takes one input.
CUBE
<- CUBE multiplies its input by itself three times.
125
← The result reported by CUBE is then input to PRINT.
PRINT

Writing simple reporters can give you your own set of personalized tools to use in other procedures. For example, you can create a pattern "tool kit" using OUTPUT so that you don't have to remember the numbers for the pen patterns.

You can write

```
TO BRICKS
OUTPUT 12
END
```

If you then type

```
PRINT BRICKS
```

you see the number 12 in the Logo Window. Using PRINT with BRICKS lets you see the pattern number. However, there is another way to use BRICKS that makes your programs more readable. You can type

```
SETPPATTERN BRICKS
SETPENSIZE 20 1
FORWARD 50
```

You see a block of bricks in the Turtle Window. You can write a series of procedures for all the patterns if you wish. You do not have to remember pattern numbers, and other people reading your program can tell what patterns you intended to use.

In a similar manner, you can write style tools. It's difficult to remember that 1 stands for bold and 4 for underline. You can create procedures for your favorite styles. If you type

```
TO BOLD.12
OUTPUT [1 12]
END
```

and then type

```
SETTSTYLE BOLD.12
```

this will allow you to easily switch to 12-point bold type.

You can save groups of procedures, such as a pattern or styles tool kit, in a separate file. When you want to use them in the workspace you are currently using, you can type

```
LOAD "file.name.with.tool.procedures
```

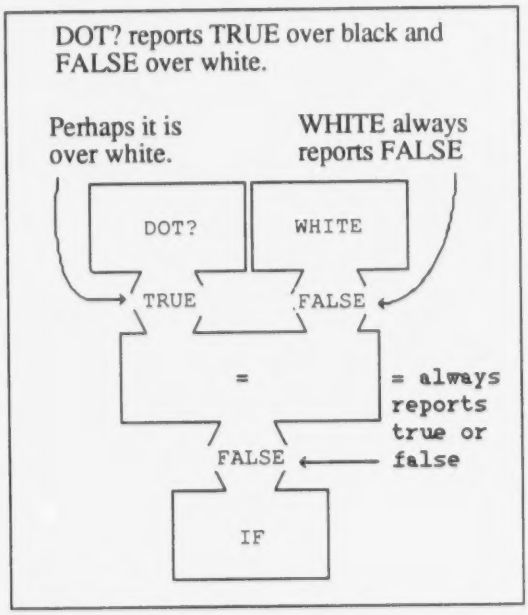
These procedures are then added to the procedures currently in memory.

Procedures such as BRICKS can be used in a variety of ways, often to increase the readability of your program. Recall the reporter DOT? that reports "TRUE" if there is a black dot under the turtle. This can be used with a tool kit to make procedures using them more readable. You might write

```
TO BLACK
OUTPUT "TRUE
END

TO WHITE
OUTPUT "FALSE
END

IF DOT? = WHITE [MOVE.AGAIN]
```



DOT? reports TRUE or FALSE. BLACK reports TRUE. WHITE reports FALSE. If the current dot under the turtle is white, then the procedure to move again is called. Perhaps you are writing a game in which you want to move until a black target is found:

```
TO MOVE.AGAIN
FORWARD 5
IF DOT? = WHITE [MOVE.AGAIN]
END
```

In the past if you wanted to write a procedure to calculate the area of a rectangle, you wrote:

```
TO AREA :LENGTH :WIDTH
PRINT :LENGTH * :WIDTH
END
```

Typing

```
AREA 8 10
```

causes Logo to respond

```
80
```

As long as you do not need to use the area elsewhere, this works fine. However, what if you want a procedure that makes use of the area, perhaps to find the cost of carpet? Then, if you write

```
TO AREA :LENGTH :WIDTH
OUTPUT :LENGTH * :WIDTH
END
```

you can use the value elsewhere. Thus, you can now type

```
PRINT 15 * AREA 8 10
```

to find the cost at $15 per square yard of an 8 yd. x 10 yd. carpet.

If you are comfortable with mathematical language, you can think of reporters as functions. That is, the AREA procedure above is a function that produces the area, given the length and width. In Logo terms, it is a procedure that takes two inputs and reports the product of those inputs. These are just different ways of saying the same thing.

Writing procedures like this as reporters allows them to be used in a number of interesting ways. Suppose you want a procedure to report the perimeter of a rectangle:

```
TO PERIMETER :LENGTH :WIDTH
OUTPUT 2 * :LENGTH + 2 * :WIDTH
END
```

You can then use this procedure to make sure the perimeter of a rectangle is within a certain range:

```
TO CHECK.PERIMETER :L :W
IF (PERIMETER :L :W) < 100→
      [(PRINT :L [and] :W [are O.K.])]→
      [(PRINT :L [and] :W [are too big.])]
END
```

This procedure can be used to plan a fenced-in area for which you had only 100 feet of fencing. Notice that the value in "L" is passed into "LENGTH" in PERIMETER and then used in PERIMETER to produce the value reported. Can you draw diagrams to explain how the reporters function and how the values are passed from input to input?

Writing your own reporters can be useful both in adding tools to your programs and in making programs more readable. When you are writing a program and discover that you want a value computed and then used elsewhere in the program, you probably need to use OUTPUT.

Tips and Techniques

There are two ways in which OUTPUT is useful at this stage of your programming career. The first is the creation of tool kits as described in this chapter. These tool kits are made up of isolated procedures that are not connected with other procedures in your program. That is, procedures in a tool kit would not appear in the procedure tree for a program. Instead each procedure in the tool kit would have its own, single-entry procedure tree.

The second use of OUTPUT is to create functions to produce some value. Writing procedures such as AREA above allow you to name a computation and then use the result of that computation anywhere in your program. In general, these functions will be a part of your procedure tree.

At this point in your Logo programming, you can think of OUTPUT as a method of naming actions or ideas in your program. You are giving a name to a color or shape number or to a computation, such as AREA. Later in your Logo programming, you will discover that OUTPUT is the key to sophisticated programming ideas using Logo.

Project Suggestions

Write your own math tool kit. Create procedures you might use frequently and put them into a file of your own tools. Be sure you put comments in your procedures so that it is clear what each procedure is for.

Use DOT? to create a simple "game." Perhaps you can have the turtle move randomly until it lands on a black area. You could have a counter that prints how many moves the turtle makes until it lands on the goal.

Develop your own style or pattern tool kit. Think carefully about naming your procedures meaningfully so that the names are not so long that they become hard to type.

32. Some Useful Graphics Reporters

Earlier you learned that Logo primitives could be divided into two categories: commands and reporters. Commands cause an immediate effect; reporters return information to Logo. In this chapter you will learn a number of reporters that can be valuable when working with *Terrapin Logo*.

When you put text in the Turtle Window by using STAMPTEXT, you are able to specify the font by using SETTFONT and the style and size of the text by using SETTSTYLE. It can be difficult to tell by looking at the Turtle Window exactly what font and size of text you are using. However, there are two reporters you can use to find out this information.

If you start Logo and then type

```
PRINT TFONT
```

you see

```
Monaco
```

If you then type

```
PRINT TSTYLE
```

you see

```
0 9
```

These results appear because the default font for *Terrapin Logo* is Monaco, normal, 9-point type. If you type

```
SETTFONT "GENEVA
SETTSTYLE [4 18]
```

you can then verify these choices easily:

```
PRINT TFONT
Geneva
PRINT TSTYLE
4 18
```

Styles are a good place to use a tool kit similar to those discussed in the last chapter. For example, you might enter

```
TO UNDERLINE
OUTPUT 4
END
```

Then it seems perfectly natural to enter

```
SETTSTYLE UNDERLINE 18
```

However, this will cause an error. Recall that SETTSTYLE requires a list of two numbers. When you use just UNDERLINE and 18, you are giving it two separate words (numbers). Your next idea might be to try

```
SETTSTYLE [UNDERLINE 18]
```

Now you get the message

```
SETTSTYLE didn't like UNDERLINE as input
It expected a number
```

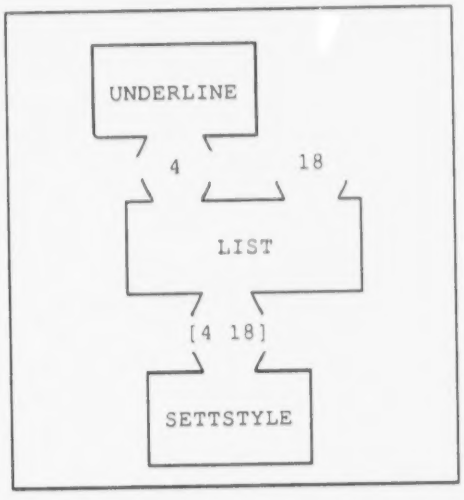
Hmmmm. But UNDERLINE represents a number, and nothing was said about a list. What is the problem?

To solve this difficulty, you have to think carefully about what you have learned about Logo grammar. The [UNDERLINE 18] is a *quoted* list. The list contains two words: UNDERLINE and 18. Recall that when information is quoted, it is *not* evaluated. So, when Logo sees UNDERLINE in a quoted list, it doesn't look for a procedure by that name. So how do we solve the problem? There is a primitive you will learn more about in a later chapter called LIST. It turns its inputs into a list, which is exactly what we wanted. So, we can write

```
SETTSTYLE LIST UNDERLINE 18
```

and the command now works as we planned. You can easily check to see that it worked with TSTYLE.

The discussion in the previous paragraph is quite technical. You may need to read it quite a few times before you understand it. Perhaps a diagram would help:



You should practice thinking carefully through the way Logo commands work. In the long run it will save you a lot of debugging time.

The previous two reporters were related to text in the Turtle Window. There is also a series of reporters related to the state of the pen. First be sure the pen is in the mode that it usually is in when you start Logo:

```
SETPPATTERN 1
SETPENSIZE 1 1
SETPENMODE 1        (OR PENPAINT)
```

Now type

```
PRINT PPATTERN
```

You see the number 1 for the current pen pattern. Try changing the pen pattern and then checking it using PPATTERN.

Next type

```
PRINT PENSIZE
```

You see

```
1 1
```

Try changing the pen size and then checking it with PENSIZE. Finally, type

```
PRINT PENMODE
```

You see a zero. Recall that zero represents the normal mode, PENPAINT. Try changing the mode using either SETPENMODE or PENMIX, PENERASE or PENREVERSE. Check using PENMODE.

Most versions of Logo have a wide variety of reporters you can use to find out information about what is happening on the screen. These can be very useful as you are working in the immediate mode. They can also be used in procedures. Suppose, for example, you had a program that allowed the user to change the style or font of the type in the Turtle Window. Perhaps you want to be sure the type is returned to Monaco font at some point in the program. You can write a command such as

```
IF NOT (TFONT = "Monaco) [SETTFONT "Monaco]
```

As you write increasingly sophisticated programs in Logo, you will find more and more uses for reporters such as those given in this chapter.

Project Suggestions

Experiment with using these new reporters. Are you sure you understand how each one works?

Write a procedure that checks the pattern and mode of the pen and returns them to "normal."

Write a procedure that checks the mode and resets the mode to PENPAINT.

33. Some Useful Text Reporters

As you might expect, *Terrapin Logo* provides you with reporters that allow you to manipulate information that appears in the Logo Window. Begin by typing

```
PRINT LOWERCASE "HELLO
```

You see

```
hello
```

Similarly, there is an UPPERCASE reporter that converts its input from mixed or lowercase letters to uppercase letters.

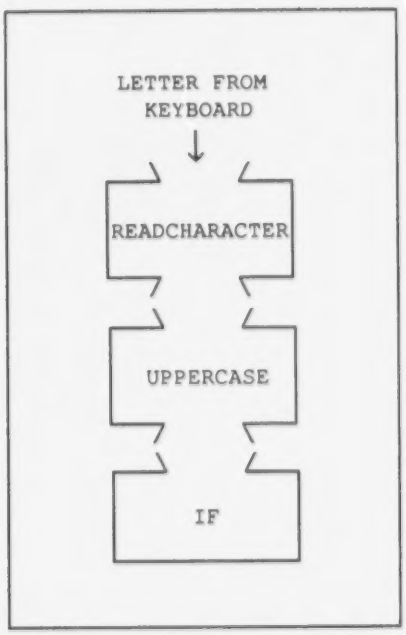
Have you ever written a command like this

```
IF READCHARACTER = "Y [GO.ON][GO.BACK]
```

only to find out that if you forget to press the Shift key and type a lowercase "Y," the GO.BACK procedure is run? UPPERCASE provides a solution for you. You can now write

```
IF UPPERCASE READCHARACTER = "Y [GO.ON][GO.BACK]
```

and be confident that if the user types a "y," then GO.ON will be run. Think about what is happening in this example.



You see that the character from the keyboard is reported to READCHARACTER. Then READCHARACTER reports that letter to UPPERCASE. Finally, UPPERCASE converts the letter to uppercase and reports the result to IF.

Two other reporters that are useful with text are ASCII and CHAR. ASCII (pronounced ass´ key) reports the ASCII value of a character. The ASCII code is the code most commonly used by computers to represent information. Each key or key combination on your keyboard has a number value associated with it. For example, type

```
PRINT ASCII "A
```

You see the number 65. Sixty-five is the ASCII value of a capital A. Try other letters. Try uppercase and lowercase. Try numbers and special symbols.

The CHAR reporter is the opposite of ASCII. Try typing

```
PRINT CHAR 65
```

You see the letter A. Try different numbers. What letters, numbers and special characters did you discover? You will find that if you use rather small numbers, nothing seems to happen. It is likely you selected numbers associated with Control characters, which don't print anything in Logo.

Earlier you saw that the CHAR reporter was useful when printing spaces. Since 32 is the ASCII value of the space bar, when you type

```
(PRINT CHAR 32 [HELLO])
```

the word HELLO is printed with one space in front of it.

The ASCII reporter can be useful in creating an "instant" program. Suppose you would like to write a procedure to allow the user to place the turtle by using the arrow keys. (Some versions of Logo include such a feature.) To begin, you must find out the ASCII value of each of the arrow keys. To accomplish this task, type

```
PRINT ASCII READCHARACTER
```

Press an arrow key and note the number. Do this for all four arrow keys. You should discover that the left arrow is 28, the right arrow is 29, the up arrow is 30, and the down arrow is 31. Thus, your procedure might look like this:

```
TO MOVE.TURTLE :ARROW
IF ASCII :ARROW = 28 [LEFT 90 FORWARD 1 RIGHT 90]
IF ASCII :ARROW = 29 [RIGHT 90 FORWARD 1 LEFT 90]
IF ASCII :ARROW = 30 [FORWARD 1]
IF ASCII :ARROW = 31 [BACK 1]
MOVE.TURTLE READCHARACTER
END
```

If you type

```
MOVE.TURTLE READCHARACTER
```

you can then move the turtle about the screen with the arrow keys. Notice that this procedure assumes that the turtle is facing the top of the Turtle Window to begin.

Project Suggestions

Modify some procedures you have written earlier that accepted answers from the keyboard so that the user no longer has to be concerned about whether s/he types upper- or lowercase.

Finish the MOVE.TURTLE program above so that the turtle begins by facing the top of the window. Add a stopping criteria so that you can leave the program without typing Command-G.

If you add the ability to raise or lower the pen to the MOVE.TURTLE program, you have a drawing utility that can be used by beginners. Experiment with various possibilities.

34. Time and the Mouse

As you no doubt know, the Macintosh has a built-in clock. *Terrapin Logo* allows you to access this information by using two reporters. If you type

```
PRINT DATE
```

you see the current date printed as three numbers separated by spaces. Actually, DATE returns a list of three numbers, but when you use the PRINT instruction the brackets are removed. Similarly, if you type

```
PRINT TIME
```

you get the time in hours, minutes, and seconds. You can use these reporters directly in PRINT instructions, for example

```
(PRINT [THE CURRENT DATE IS] DATE)
```

In the last several chapters of this book, you will learn how to manipulate lists like those produced by DATE and TIME so that you can use particular pieces, such as the hour only or the month only.

In your work with Logo, you have no doubt used the mouse frequently. Logo has several primitives for working with the mouse from a procedure. The reporter MOUSE returns a list of coordinates that give the current mouse position. This position is relative to the home position of the current turtle.

Try this. Move the turtle to its home position in the center of the Turtle Window. Put the mouse pointer on top of the turtle. Don't click the mouse button because you want the Logo Window to remain active. Now type

```
PRINT MOUSE
```

You should see two numbers that are fairly close to zero. Next, move the mouse pointer some distance from the turtle. It doesn't even have to be in the Turtle Window. Again type

```
PRINT MOUSE
```

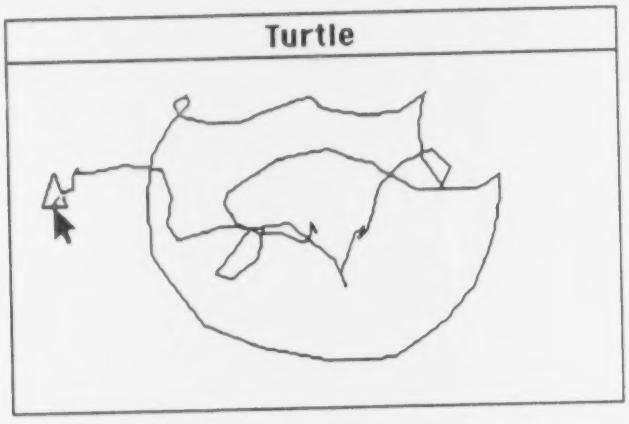
Now you should see that at least one of the numbers is some distance from zero. Experiment with this reporter.

You can write a simple procedure that makes the mouse pointer a drawing tool:

```
TO DRAW
CLEARGRAPHICS
PENDOWN
SCRIBBLE
END

TO SCRIBBLE
SETPOS MOUSE
IF BUTTON? = "FALSE [SCRIBBLE]
END
```

When you run DRAW, you can move the mouse pointer and the turtle will follow it. You will learn more about SETPOS to position the turtle in Chapter 36. BUTTON? reports TRUE if the button is down and FALSE otherwise. Thus, you only have to press the mouse button to stop the scribbling.



There are two commands available for working with the mouse, but they are potentially dangerous. If you type

```
.MOUSEOFF
```

all mouse interactions are stopped. You can only start interactions with the mouse again by typing

```
.MOUSEON
```

Note that both of these commands have a period for the first character.

Project Suggestion

Develop the mouse drawing procedure begun above. Perhaps it would be more manageable with a WAIT command. What about the ability to change pen patterns or pen widths? Could such facilities become part of a procedure that calls the SCRIBBLE procedure?

35. Using MAKE

In much that is written about Logo, you see the primitive MAKE used. Program examples in books and magazines use MAKE frequently. In fact, you may have been wondering why this book has not introduced MAKE before.

If you know another programming language, you may have been wondering why this book hasn't taught you how to assign values of variables. Indeed, the MAKE command in Logo is used for the same purpose as LET in BASIC and in the same way as the assignment operator (:=) in Pascal. MAKE is used to assign values to names.

So, why has MAKE not been introduced earlier? The primary reason is that in Logo, MAKE is frequently overused. It is often unnecessary and simply clutters both the code and the Logo system's memory. Learning to pass values among procedures is a necessary skill to learn in order to avoid overusing MAKE. Using MAKE sparingly will result in better programming style and will help you as you move into more advanced programming in Logo.

There *are* places where using MAKE is all but necessary and where it is appropriate. For example, it is not uncommon to want to increment (add to) or decrement (subtract from) a value within a Logo program. While procedure inputs can sometimes be used to accomplish this task, there are other instances where the use of MAKE is more sensible. In this chapter, you will learn the relationship between names created by MAKE and names created as procedure inputs. You will also learn how to work with names created by using MAKE.

You have used procedure inputs to associate a name with a value, for example,

```
TO GREET :WHOM
(PRINT [HI,] :WHOM)
END
```

If you type

```
GREET "MIKE
```

Logo responds

```
HI, MIKE
```

Within the procedure, you use dots (:) to see the value of the name of a procedure input. What happens if you then type

```
PRINT :WHOM
```

Logo says

```
THERE IS NO NAME WHOM
```

Why? When you typed

```
GREET "Mike
```

"WHOM" had the value "Mike" within the procedure.

The explanation lies in the fact that procedure inputs are *local* to the procedure in which they are defined. That is, both the name of the procedure input (in this case, WHOM) and the value associated with the name (in this case, Mike) cease to exist when the procedure finishes running. There are times, however, when it would be convenient to have a name retain its value throughout a program. (If you have used sequential files in another programming language, you may find that you will choose to use MAKE much as you used files elsewhere.)

Technical note: Procedure inputs are defined as long as the procedure is active. So, if a procedure calls another procedure, the name is actually accessible in that subprocedure. This is called *dynamic scoping*. If you use dynamic scoping to access the value of a name in a subprocedure, it destroys your ability to test procedures independently of other procedures. Thus it is best not to rely on it.

One place it is convenient to have a value available throughout a program is when you want to use the name of the user several times in a program. To do this, you can use a procedure like this:

```
TO ASK.NAME
PRINT [WHAT IS YOUR NAME?]
MAKE "USER READLIST
END
```

Notice the quotation mark (") in front of USER. You use the " to define a name. Then, anywhere in the program, you can use

```
PRINT :USER
```

and the value assigned to USER will be printed. We say that this name is *global* to indicate that it is understood everywhere in the program. In fact, the name and its value become a part of the workspace.

The general form of the MAKE command is

```
MAKE "some.name  some.value
```

You can read this as follows: "MAKE 'some name' have the value of 'some value.' " So, for example, if you have

```
MAKE "THE.NAMES [MARY SAM TOM SUE]
```

you read "MAKE the name THE.NAMES have the value of the list MARY, SAM, TOM, and SUE."

Since global names become a permanent part of the workspace, some commands are needed to examine and manipulate them. In order to see what names are present, you can type

```
POALL
```

and the names and their values will appear after all the procedures in your workspace are listed. If you want to see only the names and their values, you can type

```
PRINT NAMES
```

To see the value of a particular name or names, you can use

```
PONAME [list of names]
```

To remove names from memory, type

```
ERNAME [list of names]
```

Keep in mind that ERNAME will also erase any shape names you have in memory. Of course ERALL will remove the names as well as all of the procedures.

At this point you may have decided that MAKE always creates global names. Unfortunately, things are not that simple. You have seen that procedure inputs create local names, for example, in the procedure GREET:

```
TO GREET :WHOM
(PRINT [HI,] :WHOM)
END
```

You also saw how MAKE could create a global name:

```
TO ASK.NAME
PRINT [WHAT IS YOUR NAME?]
MAKE "USER READLIST
END
```

Consider this example:

```
TO CHANGE :NUMBER
PRINT :NUMBER
MAKE "NUMBER :NUMBER + 1
PRINT :NUMBER
END
```

If you run this procedure by typing

```
CHANGE 23
```

you see

```
23
24
```

in the Logo Window. Is NUMBER local to the procedure CHANGE? Try typing

```
PRINT :NUMBER
```

or

```
PRINT NAMES
```

You see that NUMBER is not a global name. It has no value outside of CHANGE. Using MAKE to change the value of a local name does not make it global.

Logo has a command that allows you to make a name that would otherwise be global into a local name. Suppose you have the procedure that we used earlier:

```
TO ASK.NAME
PRINT [WHAT IS YOUR NAME?]
MAKE "USER READLIST
END
```

If you type

```
ASK.NAME
```

and respond with

```
MARY
```

then MARY becomes the global value associated with USER. However, you can force USER to be a local name by using the LOCAL command:

```
TO ASK.NAME
LOCAL "USER
PRINT [WHAT IS YOUR NAME?]
MAKE "USER READLIST
END
```

First, erase all the names in memory using ERNAME. Then type

```
ASK.NAME
```

and respond with

```
MARY
```

Finally, type

```
PRINT NAMES
```

and you see that USER is no longer a global name.

Here are some rules of thumb to help you better understand how names work.

- When MAKE is used outside a procedure, it always creates a global name.
- When MAKE is used inside a procedure to create a new name, that name is global unless you have used LOCAL to make it into a local name.
- When MAKE is used to change the value of a procedure input (local name), that name remains local.

You should take some time to internalize these guidelines so that when you encounter MAKE, you can think about it correctly.

Tips and Techniques

As indicated at the beginning of this chapter, it is not a good idea to overuse MAKE. However, there are places where it is entirely appropriate to make use of the ability to assign global values to names. Many people, as they move into more advanced Logo, find that using a series of

MAKE commands is helpful in understanding complex Logo instructions. A simple example follows:

```
MAKE "CHOICE RANDOM 38
MAKE "PATTERN 1 + :CHOICE
SETPPATTERN :PATTERN
```

Before going any further, take a moment to examine the punctuation used with MAKE. You should read the second MAKE instruction as follows: "Make the name PATTERN have the value of 1 plus the value associated with the name CHOICE." Using MAKE is one place where most people have some difficulty with punctuation. Remember that " quotes a word and : means the value associated with a name.

Notice that the three instructions given above can be collapsed as follows:

```
MAKE "PATTERN 1 + RANDOM 38
SETPPATTERN :PATTERN
```

Or, you can just write

```
SETPPATTERN 1 + RANDOM 38
```

As this last command illustrates, neither the name PATTERN nor the name CHOICE is really needed to accomplish the task, but using several global names may be helpful in developing an understanding of the problem to be solved in writing a particular procedure. Feel free to use the MAKE command in your work to help you learn a new idea, especially in the following sections on list processing. However, when you complete a project you should attempt to "collapse" your multiple MAKE commands as much as possible. Your final work should contain only necessary (or reasonable) uses of MAKE. As your understanding of Logo programming grows, you will find this task increasingly easy.

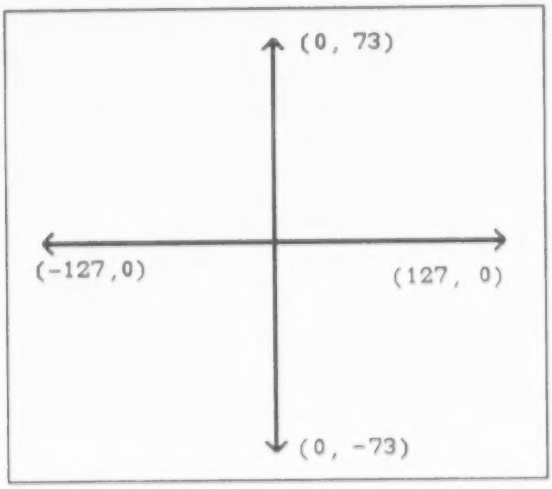
Project Suggestions

Experiment with MAKE. Be sure that you see how it works. Have you tried all of the instructions in this chapter? If not, do so. Write some short procedures using MAKE. Compare the results with the results obtained when using procedure inputs.

36. Cartesian Primitives

When you work with the turtle in the "normal" turtle graphics, you use "body centered" commands. That is, the turtle moves relative to its current position. If you say RIGHT 90, the turtle turns right from where it is currently positioned. Similarly, if you say FORWARD 50, the turtle moves forward from its current position without changing direction.

Sometimes it is useful to be able to move the turtle to a particular position in the Turtle Window from within a procedure. It is also useful to be able to have the turtle face a particular direction. The commands that do this are based on graphing, similar to the graphing you may have done in math class. You can think of the Turtle Window as a sheet of graph paper:

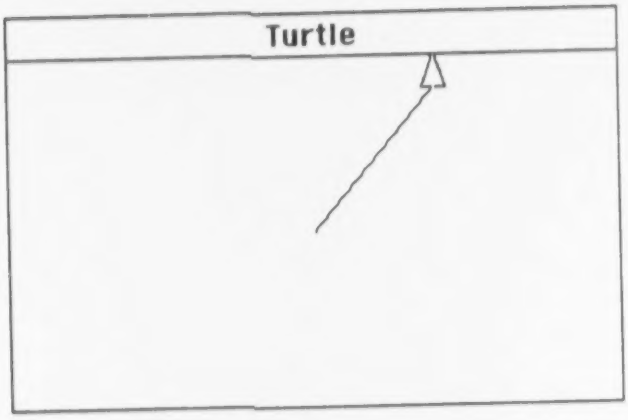


Notice that the origin, the point (0, 0), is in the middle of the screen where the turtle is found when you start Logo. The coordinates given in the diagram above only apply to the Turtle Window as it appears when you first start Logo. If you change the size of the turtle window, then the coordinates will be different.

To place the turtle at a particular position in the Turtle Window, recall that you can type

```
SETXY 50 60
```

The turtle moves 50 steps to the right of the center of the screen and 60 steps up. If the pen is down, the turtle leaves a trail.



There are two reporters that allow you to see the coordinates of the turtle at any time. They are XCOR and YCOR. If you type

```
XCOR
```

you see

```
RESULT: 50
```

Similarly, typing

```
YCOR
```

produces

```
RESULT: 60
```

Terrapin Logo has a second primitive that has the same effect as SETXY, but it works slightly differently. If you type

```
CLEARGRAPHICS
SETPOS [50 60]
```

the Turtle Window will look exactly as it does above. Note that SETXY needs two numbers as input while SETPOS needs a *list* of two numbers as input. In your work with Logo, you will sometimes find one more convenient to use than the other. There is a reporter associated with SETPOS that returns both the x- and y-coordinates of the turtle. Thus, if you type

```
POS
```

you see

```
Result: [50 60]
```

These commands can also be used to create a point-graphing program. For example, you might type

```
TO GRAPH.POINT.
PRINT [WHAT POINT DO YOU WANT TO GRAPH?]
PENUP
SETXY READWORD READWORD
PENDOWN
FORWARD 1
PRINT [ANOTHER?]
IF READCHARACTER = "Y [GRAPH.POINT]
END
```

To use this program, you would probably want a SET.UP program that puts axes in the Turtle Window and sets the pen pattern and size.

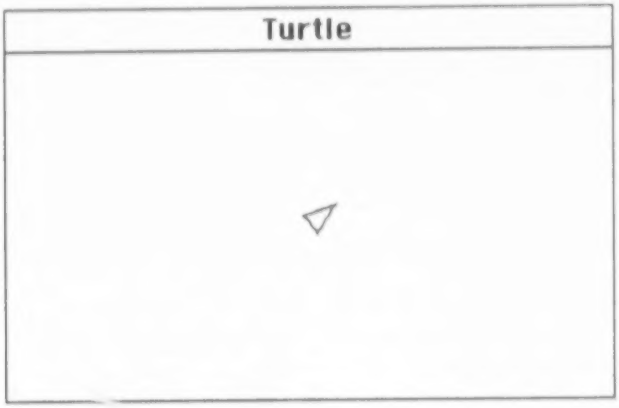
You can also specify the direction the turtle faces. Type

```
SETHEADING 45
```

or

```
SETH 45
```

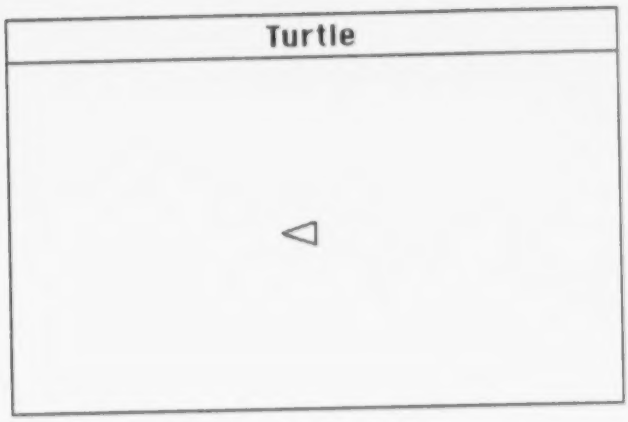
and the turtle turns to face the upper right corner of the Turtle Window.



Type

```
SETH 270
```

and the turtle faces the left side of the screen.



You can also check the direction the turtle is heading by typing

```
PRINT HEADING
```

The current direction the turtle is facing appears on the text screen. If you type

```
SETHEADING 117
PRINT HEADING
```

you see

```
117
```

Cartesian commands and reporters can be useful in a variety of programs. Suppose you are writing a game program, and you want to know if the turtle is over a colored target. Recall that the reporter DOT? returns TRUE if there is a dot under the turtle and FALSE if there isn't. Thus, you might have

```
IF DOT? = "TRUE [PRINT [YOU WIN!]]
```

Terrapin Logo also provides you with a very convenient DISTANCE primitive. DISTANCE needs two inputs, the coordinates of a point. It then returns the distance from the turtle to that point. Thus, if you type

```
PRINT DISTANCE 0 60
```

you see

```
60
```

since the turtle would only need a FORWARD 60 to get to that point. Similarly,

```
PRINT DISTANCE 50 0
```

gives

```
50
```

Of course, you can get the distance to any point in the Turtle Window. If you type

```
PRINT DISTANCE 50 60
```

you see

```
78.102496759067
```

which is the number of turtle steps needed to get from the center of the screen to the point (50, 60).

Terrapin Logo also provides the primitive DISTANCEPOS. It works exactly like DISTANCE except that it requires a list of two numbers as input. Like SETXY and SETPOS, sometimes you will want to use one of these and sometimes the other.

Finally, there is the powerful reporter TOWARDS. TOWARDS returns the heading of the turtle if it faces a particular point. If you type

```
CLEARGRAPHICS
TOWARDS 50 50
```

Logo prints

```
Result: 45
```

This means that the heading of the turtle when it is in its current location and is turned to face the point (50, 50) will be 45. Most often, TOWARDS is used to turn the turtle a particular direction, as in

```
SETHEADING TOWARDS 50 50
```

Again, this is a primitive that is useful in writing games.

Tips and Techniques

Earlier in this book you saw that some of these Cartesian reporters could be helpful in your work with "standard" turtle graphics. For example, you could place the turtle and then type

```
(PRINT XCOR YCOR)
```

to determine its location so that you could use SETXY in a procedure. Often these Cartesian reporters can be very helpful in debugging programs. Putting instructions such as

```
(PRINT XCOR YCOR)
PRINT HEADING
```

in a procedure in which the turtle is moving gives you a running account of what is happening. This can be very helpful in an animation program.

In general, reporters that return some information about the turtle or the cursor are good tools to use when trying to determine exactly what your program is doing. Keep in mind that instructions such as

```
PRINT XCOR
PRINT YCOR
PRINT DOT?
PRINT TOWARDS
```

are valuable tools that let Logo help you debug your programs.

Project Suggestions

Practice using these coordinate commands. Be sure you understand how each works. Write some procedures using coordinate commands to draw shapes.

Complete the graphing program started above. Add labels to the axes. Put "tic marks" on the axes and label them as well.

Create a game program. Decide on a target. Test to see if the turtle has landed on the target.

37. Predicates

Predicate is a new term used to categorize procedures. It refers to a special *kind* of reporter. You have seen and used quite a few reporters. Different reporters report different kinds of Logo objects. For example, RANDOM reports numbers while DOT? returns TRUE or FALSE. The term *predicate* refers to reporters that return *only* TRUE or FALSE. You have encountered several of these special reporters earlier in this book.

Why would you need such a class of reporters? Think about what goes after IF. You have always used a condition containing =, <, or >. Earlier you saw that these symbols are actually reporters. Since they return only TRUE or FALSE, they belong to the special class of reporters called predicates. For example,

```
PRINT 3 = 4
```

displays

```
FALSE
```

while

```
PRINT 3 < 4
```

displays

```
TRUE
```

Most of the new reporters you will learn about in this chapter end in a "?" (In many versions of Logo, predicates end in "p" for predicate.) There is nothing magic about the "?" at the end. It is just there to alert you that you are working with a predicate. Putting a "?" at the end of a procedure name does NOT turn that procedure into a predicate. Keep in mind that what makes a procedure a predicate is the fact that it reports *only* TRUE or FALSE.

There are three predicates used with IF to create compound conditions. These reporters are NOT, AND, and OR.

NOT reverses the output from whatever follows it:

```
PRINT 3 = 4
FALSE
PRINT NOT 3 = 4
TRUE
```

AND and OR are used when you want to examine more than one condition. AND returns TRUE when all the conditions after it are true and returns FALSE otherwise. OR returns TRUE when any of the conditions after it are true; it returns TRUE if they are all false:

```
PRINT 5 > 0
TRUE
PRINT 5 < 10
TRUE
PRINT AND (5 > 0) (5 < 10)
TRUE
```

```
PRINT 5 > 0
TRUE
PRINT 5 > 10
FALSE
PRINT OR (5 > 0) (5 > 10)
TRUE
```

It takes practice to use these three reporters. Once you master them, you can create more complex IF instructions rather than numerous simple ones.

You might use AND to check that a number was in the proper input range, for example,

```
TO CHECK :NUMBER
IF AND (:NUMBER > 0) (:NUMBER < 100)→
      [PRINT [YOU ARE IN THE CORRECT RANGE.]]→
      [PRINT [PICK A NUMBER BETWEEN 0 AND 100.]]
END
```

OR can be useful if you want to write a game that controls the range of the turtle, for example,

```
IF OR (XCOR > 125) (XCOR < -125) [RIGHT 180]
```

The following instruction will bounce the turtle off the sides of the standard Turtle Window:

```
IF (OR ((XCOR) > 125)→
       ((XCOR) < -125)→
       ((YCOR) > 70)→
       ((YCOR) < -70)) [RIGHT 180]
```

If the turtle reaches the edge of the window, it turns completely around.

There is yet another group of reporters. These are useful for checking input:

NUMBER?	returns TRUE if its input is a number, otherwise FALSE.
WORD?	returns TRUE if its input is a word, otherwise FALSE.
LIST?	returns TRUE if its input is a list, otherwise FALSE.

For example, you might write a procedure like this:

```
TO DRAW.PICTURE
PRINT [TYPE A NUMBER:]
CHECK.AND.DRAW READWORD
END

TO CHECK.AND.DRAW :INPUT
IF NUMBER? :INPUT [FORWARD :INPUT] [DRAW.PICTURE]
END
```

When you type DRAW.PICTURE, Logo prints

```
TYPE A NUMBER:
```

If you type a number, the turtle moves forward. If you type anything else, the procedure asks for a number again.

When you are working with words and lists, you will find that it is not uncommon to need to work with a word or list with nothing in it. The predicate EMPTY? allows you to check to see if there is anything in a Logo object. For example, you might have

```
IF EMPTY? :SOME.OBJECT [PRINT [NO MORE]] [PRINT [SOMETHING LEFT]]
```

You will find this predicate useful as you learn about list processing in the last few chapters of this book.

Another useful predicate that returns TRUE or FALSE is MEMBER? MEMBER? checks to see if its first input is contained in its second input. If so, it returns TRUE. Otherwise it reports FALSE. For example, if you want to check to see if a character is a vowel, you can use MEMBER?:

```
TO CHECK :CHARACTER
IF MEMBER? :CHARACTER [A E I O U]→
      [PRINT [A VOWEL]]→
      [PRINT [NOT A VOWEL]]
END
```

If you type

```
CHECK "A
```

Logo displays

```
A VOWEL
```

but if you type

```
CHECK "B
```

Logo displays

```
NOT A VOWEL
```

You have already seen that BUTTON? is useful when working with the mouse. BUTTON? returns TRUE when the mouse button is down, FALSE otherwise. Perhaps you have noted that in earlier chapters when a predicate was used, the IF instruction was written

```
IF BUTTON? = "FALSE [MOVE.TURTLE]
```

However, in this chapter, you see that the second part of the condition is not necessary. BUTTON? can return TRUE or FALSE directly to the IF command without the use of the equal sign (=). Thus you can write

```
IF BUTTON? [MOVE.TURTLE]
```

A predicate that is useful when working with the turtle is PENDOWN? PENDOWN? returns TRUE if the pen is down and FALSE otherwise. This could be used to be certain that the pen really is down before drawing begins:

```
IF NOT PENDOWN? [PENDOWN]
```

Finally, there is a predicate that is useful when working with text. Type

```
PRINT BEFORE? "CAR "CAT
```

Logo prints

```
TRUE
```

BEFORE? can be useful for putting Logo words in ASCII order.

Tips and Techniques

Some versions of Logo also have GREATER? and LESS? primitives. *Terrapin Logo* does not, but you can write them as follows:

```
TO GREATER? :INPUT.1 :INPUT.2
OUTPUT :INPUT.1 > :INPUT.2
END

TO LESS? :INPUT.1 :INPUT.2
OUTPUT :INPUT.1 < :INPUT.2
END
```

Using these predicates will avoid the problems sometimes created when you mix the infix notation of standard mathematics with Logo's prefix notation.

The predicates AND, OR, and NOT are very powerful but can be confusing to use at first. Recall that one technique for testing conditions is to run them in the immediate mode. So, for example, if you have the instruction

```
IF AND (:NUMBER > 0) (:NUMBER < 100)
```

and it doesn't seem to be working correctly, you can test it by running it with specific numbers. For example, you might test

```
AND (5 > 0) (5 < 100)
AND (150 > 0) (150 < 100)
```

Studying the results of such tests will help you understand AND, OR, and NOT better, as well as help you debug your programs.

Writing your own predicates can be a way of simplifying your programs. Thus, if you have an instruction such as

```
IF (OR ((XCOR) > 125)→
       ((XCOR) < -125)→
       ((YCOR) > 70)→
       ((YCOR) < -70)) [RIGHT 180]
```

you can move the complex part of this instruction to the procedure

```
TO EDGE?
OUTPUT (OR ((XCOR) > 125) ((XCOR) < -125) ((YCOR) > 70) ((YCOR) < -70))
END
```

and the IF instruction then becomes

```
IF EDGE? [RIGHT 180]
```

making your program much easier to read and debug.

Project Suggestions

Practice creating compound conditions using AND, OR, and NOT. Simply use PRINT instructions to check the results of these conditions. The more you practice using these reporters, the easier it will be when you need them in a program. Can you write a procedure to check for a range of numbers of your choice? Can you keep the turtle within a "track" in the Turtle Window? Where would NOT be useful?

Put DRAW.PICTURE into a drawing utility in which the user can input both distance and angle. Be sure to check that the user types a number for the angle.

38. FIRST, BUTFIRST, LAST, and BUTLAST

You have worked with text primarily using PRINT and interactive reporters. There are a lot more reporters that allow you to manipulate text. You have seen FIRST when you wrote a READWORD procedure. FIRST is one of the primitives in Logo that is used with what is traditionally called *list processing*. List processing involves working with Logo objects associated with local or global names. All the primitives you will examine in this and the following three sections are reporters. Even though they are list-processing primitives, they can be used to manipulate any kind of Logo object: numbers, words, or lists.

The earliest versions of Logo did not contain a turtle at all. They had none of the elaborate features of modern versions of Logo. List processing was at the center of any activities that were done with Logo. However, writing list-processing programs is not always easy. It takes programming skill, understanding of reporters, and familiarity with recursion, including embedded recursion. These next several chapters merely introduce you to the primitives needed to do sophisticated list processing. If you are interested in pursuing these ideas, then you should move next to Brian Harvey's *Computer Science Logo Style, Volume 1*, mentioned earlier in this book.

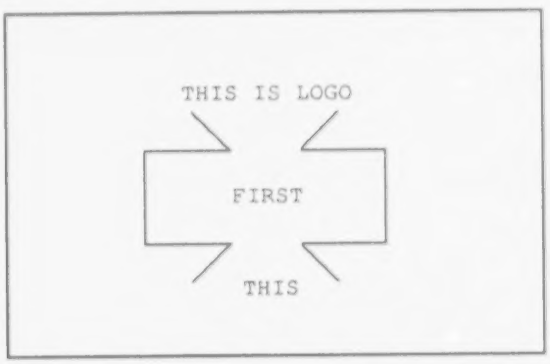
Begin by studying these simple examples:

```
PRINT FIRST "LOGO
L

PRINT FIRST [THIS IS LOGO]
THIS

PRINT FIRST 54321
5
```

It should be clear that the reporter FIRST takes either a word or a list as input (a number is a special kind of word) and returns the first letter in a word or the first element in a list. We have diagrammed such reporters like this:



To be sure you understand FIRST, what do you think Logo displays if you type

```
PRINT FIRST [[LIST ONE] [TWO] [AND THREE]]
```

Did you correctly deduce that Logo displays the first element of the list, which is the list [LIST ONE]?

There is also a reporter LAST. What do you think it does? What input does it need? What output does it produce? Can you state your thoughts clearly and precisely? Take a few moments to experiment with LAST.

There are two companions to FIRST and LAST. Study these examples:

```
PRINT BUTFIRST "LOGO
OGO

PRINT BUTFIRST [THIS IS LOGO]
IS LOGO

PRINT BUTFIRST 54321
4321
```

If BUTFIRST (short form BF) gets a word as input, it returns everything but the first letter of the word; similarly, if it gets a list as input, it returns everything but the first element of the list.

As you might guess, there is also a BUTLAST reporter. BL is the short form. Can you describe precisely how it works? Can you predict what examples like the ones above will produce? Can you diagram the way it works? Experiment with BUTLAST.

These two pairs of reporters can be used to take lists apart using recursion, much as you incremented numerical values earlier in this book. Consider this procedure:

```
TO TAKE.APART :LIST.IN
PRINT FIRST :LIST.IN
TAKE.APART BUTFIRST :LIST.IN
END
```

If you type

```
TAKE.APART [THIS IS LOGO]
```

Logo prints

```
THIS
IS
LOGO
```

in the Logo Window and displays

```
FIRST didn't like [] as input.
It expected a list, word, or string with at least one element
at level 4 in line
PRINT FIRST :LIST.IN
of TAKE.APART
```

FIRST will not accept an empty word or list as input. To avoid this problem and stop a procedure such as this, you can modify the last line as follows:

```
TO TAKE.APART :LIST.IN
PRINT FIRST :LIST.IN
IF NOT EMPTY? BUTFIRST :LIST.IN→
      [TAKE.APART BUTFIRST :LIST.IN]
END
```

Recall that EMPTY? is a predicate that returns TRUE if the word or list following it has nothing in it; it returns FALSE otherwise. Do you see why you must have the BUTFIRST in the last line rather than just :LIST.IN? Examine the diagram below to see what is happening. It shows the changing values of the procedure input instead of the name of the procedure input.

```
TO TAKE.APART [THIS IS LOGO]
PRINT [THIS]
IF NOT EMPTY? [IS LOGO]
       [TAKE.APART[IS LOGO]]
END
              ↘
               TO TAKE.APART [IS LOGO]
               PRINT [IS]
               IF NOT EMPTY? [IS LOGO]
                      [TAKE.APART [LOGO]]
               END
                            ↘
                             TO TAKE.APART [LOGO]
                             PRINT [LOGO]
                             IF NOT EMPTY? [ ] → the procedure stops
                                    [TAKE.APART []]
                             END
```

Take some time to be sure you understand what occurs in this procedure.

Tips and Techniques

You will often see a procedure like TAKE.APART written as follows:

```
TO TAKE.APART :LIST.IN
IF EMPTY? :LIST.IN [STOP]
PRINT FIRST :LIST.IN
TAKE.APART BUTFIRST :LIST.IN
END
```

This second version is a little easier to understand. It is not complicated by passing BUTFIRST :LIST.IN in TAKE.APART. However, you need to be careful when you use this form that you don't overuse STOP. It is easy to find yourself with a procedure such as this with several STOPs in it and even some unwanted embedded recursion.

Thoroughly understanding the simple ideas presented in this chapter is the first step to moving into more advanced Logo programming. Beware, however, of the simplicity of the ideas. For example, a common problem is to not think carefully about whether you are working with a Logo word or a Logo list. As you begin your work with list processing, it becomes very important to pay close attention to the sometimes subtle distinction between Logo words and Logo lists.

Project Suggestions

Spend some time experimenting with each of these new reporters as suggested above. Be sure you understand how each functions. Pay particular attention to the difference between words and lists.

Diagram the TAKE.APART procedure using a word as input instead of a list.

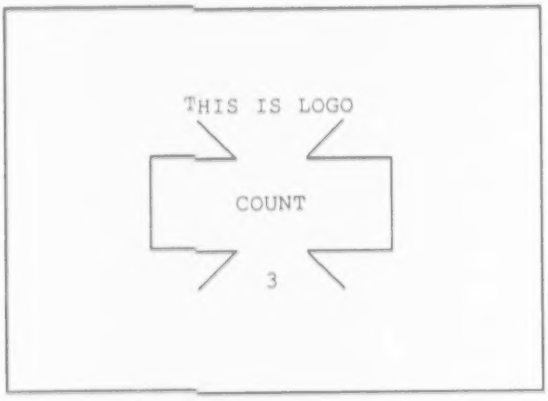
Modify the TAKE.APART procedure so that it lists the words in reverse order.

39. ITEM and COUNT

You learned how to increment using recursion. Until now, that was the only way to count things in Logo. There is a list-processing reporter that counts the objects in any word or list. Study these examples:

```
PRINT COUNT "LOGO
4

PRINT COUNT [THIS IS LOGO]
3
```




```
PRINT COUNT [A [B C] [D] [E F G]]
4
```

COUNT takes a word or list as input and returns the number of letters in the word or the number of elements in the list. In the last example, there are four elements in the list—the word A, and the lists [B C], [D], and [E F G]. Each counts as one Logo object in the list.

To best examine the next reporter, you first need to know about FPRINT. Examine these examples:

```
PRINT "HI
HI

FPRINT "HI
HI

PRINT [HI]
HI

FPRINT [HI]
[HI]
```

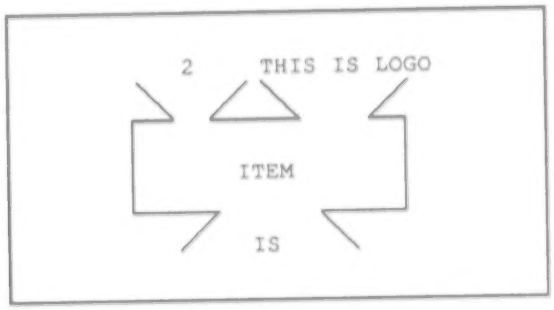
Notice that there is no apparent difference between the way PRINT and FPRINT work with words. However, you can see that PRINT removes the outer brackets from a list while FPRINT

does not. There are some other technical differences between these two commands, but the one most important now is the fact that we can use FPRINT to see if the result of an instruction is a word or a list.

Yet another reporter allows you to examine a particular letter in a word or object in a list. Consider these examples:

```
FPRINT ITEM 3 "LOGO
G

FPRINT ITEM 2 [THIS IS LOGO]
IS
```




```
FPRINT ITEM 3 [A [B C] [D] [E F G]]
[D]
```

ITEM takes two inputs: a number and either a word or a list. It returns the letter in the word or the element in the list in the place indicated by the number.

These new primitives can be put together to select a random element from a list. First, consider this instruction:

```
PRINT RANDOM COUNT [THIS IS LOGO]
```

The possible numbers printed are 0, 1, and 2. Do you see why? (RANDOM reports numbers from 0 to 1 less than its input.) To pick randomly from a list, you can't use 0; ITEM needs a number between 1 and the number of objects in its input. This can be accomplished by using

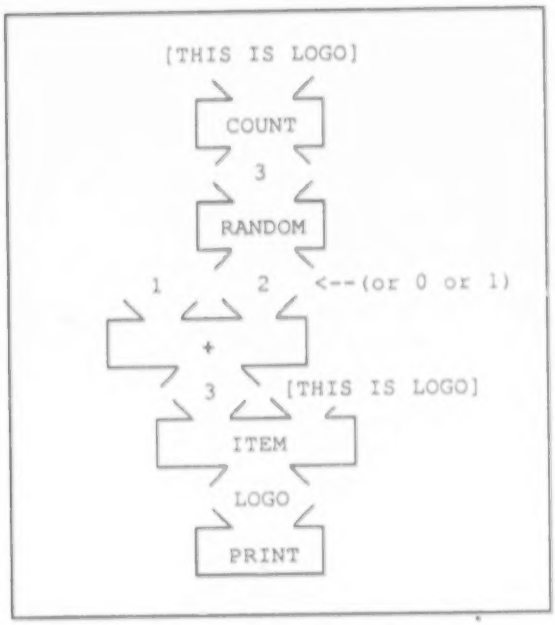
```
PRINT 1 + RANDOM COUNT [THIS IS LOGO]
```

This gives the numbers 1, 2, or 3.

Next, the problem is to pick the element represented by this number. ITEM should accomplish this:

```
PRINT ITEM (1 + RANDOM COUNT [THIS IS LOGO]) [THIS IS LOGO]
```

Examine the instruction above carefully. Study the diagram below. Look at one "box" at a time to see what is happening. Learning to think through complex instructions like this is one of the keys in learning to use list-processing reporters in Logo.



As it stands, the above example is not particularly useful. However, the idea can be put into a procedure that accepts an input:

```
TO PICK :IN
PRINT ITEM (1 + RANDOM COUNT :IN) :IN
END
```

If you type

```
PICK "LOGO
```

then one of the four letters is printed, and if you type

```
PICK [THIS IS LOGO]
```

then one of the three words in the list is printed, as shown in the above diagram.

However, to make PICK a tool that is useful in other procedures, the PRINT needs to be changed to OUTPUT:

```
TO PICK :IN
OUTPUT ITEM (1 + RANDOM COUNT :IN) :IN
END
```

Now, typing

```
PRINT PICK [THIS IS LOGO]
```

causes one of the three elements in the list to be printed in the Logo Window.

PICK can be used as a tool to generate random sentences. Suppose that you write the following program:

```
TO SILLY.SENTENCES
PRINT [TYPE SEVERAL NOUNS AND THEN PRESS RETURN]
MAKE "NOUNS READLIST
PRINT [TYPE SEVERAL VERBS AND THEN PRESS RETURN]
MAKE "VERBS READLIST
(PRINT PICK :NOUNS PICK :VERBS)
END
```

Logo now writes a random sentence using the words you entered:

```
Dog ran
```

or

```
House jumped
```

or

```
Tree fell
```

and so forth.

Tips and Techniques

This PICK procedure is such a useful tool that many feel it should be a primitive in Logo. In fact, the name PICK is used for this procedure in much that is written about Logo. With PICK you get the first real hint of Logo's extensibility. If a version of Logo doesn't have a particular primitive (XCOR, GREATER?, PICK, etc.) you can often just write it and add it to your workspace when needed. If you explore Logo programming in more depth, you will find yourself writing procedures that behave much like primitives. These procedures become your own personal "tool kit."

Project Suggestions

Experiment with ITEM and COUNT. Can you find ways to use them in programs you want to write? Be sure you understand how these new reporters work.

Expand on the above sentence generator. Can you add articles, adjectives, and adverbs?

Draw a diagram using inputs different from "THIS IS LOGO" for the line PRINT ITEM (1 + RANDOM COUNT :IN) :IN.

Write a program using PICK to deal a hand of cards. You might use a very small, incomplete deck of cards as you create your program, or you might have your program pick from two lists, one containing the suits and one containing the ranks (Ace - King).

Can you write a MY.FIRST procedure that acts like FIRST but doesn't use FIRST? How about a MY.BUTFIRST?

Here's a challenge: Write a MY.ITEM procedure that behaves just like ITEM but doesn't use ITEM. Can you write a MY.COUNT?

40. WORD, SENTENCE, and LIST

So far you have used the list-processing primitives to take lists and words apart, or to find one part of a list or word. There are times when you might also want to put words and/or lists together. This next group of reporters is used to accomplish this task.

Suppose you want to add an S to a word to make it plural. You might try typing

```
(PRINT "DOG "S)
```

but Logo displays

```
DOG S
```

You can solve the problem with the reporter WORD. Typing

```
PRINT WORD "DOG "S
```

causes the word

```
DOGS
```

to be displayed. WORD takes two words as input and returns a single word.

The related reporter SENTENCE (short form SE) accepts two words or lists as inputs and returns a single list. Thus, if you type

```
PRINT SENTENCE [THESE ARE] [TWO PARTS]
```

Logo displays

```
THESE ARE TWO PARTS
```

Recall that FPRINT displays the brackets when it prints a list. This property of FPRINT can be very helpful when working with the list-processing capabilities of Logo. If you type

```
FPRINT SENTENCE [THESE ARE] [TWO PARTS]
```

then you see

```
[THESE ARE TWO PARTS]
```

PRINT removes the outer pair of brackets.

In some Logo books, you will often see instructions like this:

```
PRINT SENTENCE [HI,] :NAME
```

instead of

```
(PRINT [HI,] :NAME)
```

Many early versions of Logo did not allow you to use parentheses to group inputs to PRINT, so you had to use SENTENCE to create a single input for PRINT.

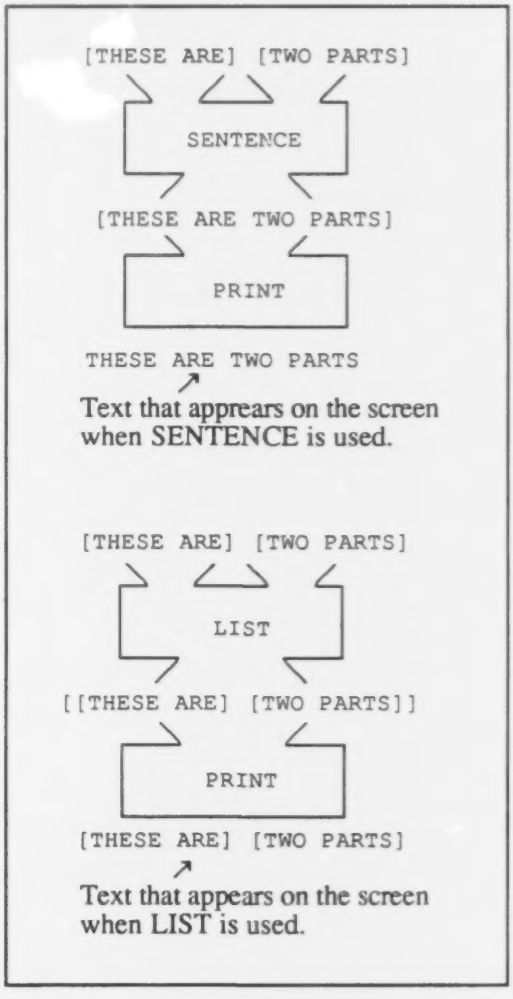
LIST is another reporter that accepts two words or lists as input and returns a list. For example, if you type

```
FPRINT LIST [THESE ARE] [TWO PARTS]
```

Logo displays

```
[[THESE ARE][TWO PARTS]]
```

Compare SENTENCE and LIST. Notice the difference in output with the same input. You may find the differences to be a bit confusing. The secret lies in the fact that SENTENCE "flattens" a list. It removes extra brackets. Examine the diagram below. Do you see the difference?


[THESE ARE] [TWO PARTS]
SENTENCE
[THESE ARE TWO PARTS]
PRINT
THESE ARE TWO PARTS
Text that apprears on the screen when SENTENCE is used.
[THESE ARE] [TWO PARTS]
LIST
[[THESE ARE] [TWO PARTS]]
PRINT
[THESE ARE] [TWO PARTS]
Text that appears on the screen when LIST is used.

This technical distinction sometimes becomes important in more advanced work with lists.

One way you might use SENTENCE is to create a list from words typed at the keyboard:

```
TO GET.WORDS :THE.LIST
PRINT [TYPE A WORD.  TYPE S TO STOP.]
MAKE "THE.LIST SENTENCE :THE.LIST READWORD
IF (LAST :THE.LIST) = "S→
      [USE.LIST :THE.LIST STOP]→
      [GET.WORDS :THE.LIST]
END
```

USE.LIST is a procedure (not shown) that would then make use of the list created in GET.WORDS.

The diagram below follows the procedure GET.WORDS through several steps using the values for the name THE.LIST.

```
TO GET.WORDS [ ]
PRINT [TYPE a word.. Type S to stop.]
MAKE "THE.LIST SENTENCE [ ] READWORD ←—— Cat is typed
IF (LAST [Cat] = "S
   [USE.LIST [Cat] STOP]] ←—— This line is not run.
   [GET.WORDS [Cat]]
END
```

```
TO GET.WORDS [Cat]
PRINT [TYPE a word.. Type S to stop.]
MAKE "THE.LIST SENTENCE [Cat] READWORD ←—— Dog is typed
IF (LAST [Cat Dog] = "S
   [USE.LIST [Cat dog] STOP]] ←—— This line is not run.
   [GET.WORDS [Cat Dog]]
END
```

```
TO GET.WORDS [Cat Dog]
PRINT [TYPE a word.. Type S to stop.]
MAKE "THE.LIST SENTENCE [Cat Dog] READWORD ←—— S is typed
IF (LAST [Cat Dog] S]) = "S
   [USE.LIST [Cat Dog S] STOP]
   [GET.WORDS [Cat Dog]] ←—— Now this line is not run.
END
```

As you looked at the diagram, did you see the bug in this program? Look carefully at the last call to USE.LIST. Do you see that the S is being passed to USE.LIST as one of the words in the list? Can you fix the bug?

Project Suggestions

Take the diagram of GET.WORDS and go one step further. That is, show the value of LAST :THE.LIST instead of just :THE.LIST.

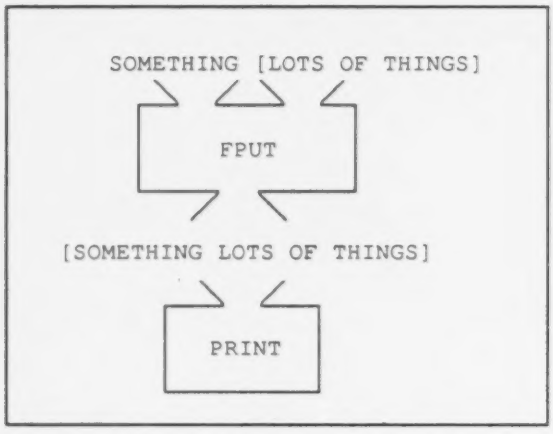
Take some time to explore the differences among WORD, SENTENCE, and LIST. Use FPRINT with both simple and compound lists. See if you can predict the outcome of each instruction before you press Enter.

Use the ideas for creating random sentences and the ideas from this chapter for making lists of words and write a program that will accept lists of words from the keyboard and print random sentences in the Logo Window.

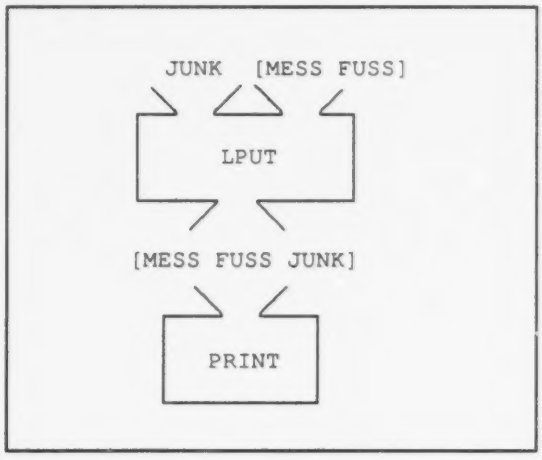
41. FPUT and LPUT

Sometimes you want to add things to an existing list rather than just put lists together. The primitives to accomplish this task are LPUT and FPUT. Here are a couple of examples:

```
FPRINT FPUT "SOMETHING [LOTS OF THINGS]
[SOMETHING LOTS OF THINGS]
```




```
FPRINT LPUT "JUNK [MESS FUSS]
[MESS FUSS JUNK]
```



FPUT accepts an object and a list as input and reports a list made up of the object and the list. LPUT accepts an object and a list as input and reports a list made up of the list with the first object on the end.

An example in the last chapter showed a way to create a list of words. Here is another approach, this time not using MAKE:

```
TO ADD.NOUNS :LIST.IN
PRINT []
PRINT1 [(NEXT) NOUN ->]
ADD.MORE READWORD :LIST.IN
END

TO ADD.MORE :WORD.IN :THE.LIST
IF NOT (:WORD.IN = "QUIT )→
      [ADD.NOUNS FPUT :WORD.IN :THE.LIST]
END
```

The second procedure is needed to "hold on" to the value typed so that it can be checked. The two procedures work together to produce a list that can be used. Below is an example of running this procedure:

```
ADD.NOUNS []

(NEXT) NOUN ->CAT
(NEXT) NOUN ->DOG
(NEXT) NOUN ->RAT
(NEXT) NOUN ->QUIT
```

As with the other list-processing primitives, you should spend some time working with FPUT and LPUT. Be sure you understand how they work. FPRINT is a good vehicle for being sure whether you are working with words or lists.

Tips and Techniques

You have now seen the list-processing primitives necessary for advanced programming in Logo. Have you noticed that they fall into several categories?

FIRST, LAST, BUTFIRST, BUTLAST: used to take Logo objects apart
ITEM, COUNT: used to examine Logo objects
WORD, SENTENCE, LIST: used to put Logo objects together
FPUT, LPUT: used to add Logo objects to a list

Keeping these categories in mind is helpful as you think about writing a program that manipulates Logo objects.

Project Suggestion

Draw a diagram showing what happens when ADD.NOUNS is run with the words shown in the sample run in the text.

Quick Reference Card

(Adapted from the *Terrapin Logo for the Macintosh: Tutorial and Reference Guide*)

command or reporter (operation)
* reporter (operation)
< > optional inputs

Graphics Commands

```
# ASK turtlenumber [ list of commands
      ]
  BACK (BK) number
  CLEAN
  CLEARGRAPHICS (CG)
* DISTANCE number1 number2
* DISTANCEPOS
      [ number1 number2 ]
* DOT?
  EACH [ list of commands ]
  FILLSH [ list of commands ]
  FORWARD (FD) number
* HEADING
  HIDETURTLE (HT)
  HOME
* INRECT? [ number1 number2 ]
    [ number3 number4 number5
    number6 ]
  LEFT (LT) number
  NOWRAP
  PENDOWN (PD)
* PENDOWN?
  PENERASE (PE)
  PENMIX
* PENMODE
  PENPAINT
  PENREVERSE (PX)
* PENSIZE
  PENUP (PU)
* POS
* PPATTERN
  RIGHT (RT) number
  SETHEADING (SETH) number
  SETNTURTLES number
  SETPENMODE numbe.
  SETPENSIZE (SETPS) number1 number2
  SETPOS [ number1 number2 ]
  SETPPATTERN (SETPP) number
  SETSHAPE array (or snaplist)
  SETTFONT word/string
  SETTSTYLE [ word number ]
  SETTWINDOW word/string
  SETX number
  SETXY number1 number2
  SETY number
* SHAPE
  SHOWTURTLE (ST)
  SNAP number1 number2
  STAMP snaplist <number1 number2>
  STAMPARC number1 number2 number3
      <predicate>
  STAMPOVAL number1 number2
      <predicate>
  STAMPRECT number1 number2
      <predicate>
  STAMPRRECT number1 number2 number3
      <predicate>
  STAMPTEXT word/string number1
      number2 <number>
  TELL word/list
* TFONT
* TOWARDS number1 number2
* TOWARDSPOS [ number1 number2 ]
* TSTYLE
* TURTLES
* TWINDOW
* WHO
  WRAP
* XCOR
* YCOR
```

Array-Handling Primitives

```
* AGET array list
* ARRAY number/list <list>
* ARRAY? object
* ARRAYDIMS array
  ASET array number/list object
* BYTEARRAY number <list>
* BYTEARRAY? array
  FILLARRAY array list
  LISTARRAY array
```

Naming

```
  LOCAL word1 <word2 word3 … >
  MAKE word1 word2
  THING word
* THING? object
```

Numeric Reporters (Operations)

* + *number1 number2 (infix)*
* - *number1 number2 (infix)*
* * *number1 number2 (infix)*
* / *number1 number2 (infix)*
* > *number1 number2 (infix)*
* < *number1 number2 (infix)*
* ABS *number*
* ATAN *number1 number2*
* BITAND *number1 number2*
* BITOR *number1 number2*
* BITXOR *number1 number2*
* COS *number*
* DIVIDE *number1 number2*
* EXP *number*
* FLOOR *number*
* INTEGER *number*
* LN *number*
* MINUS *number1 number2*
* PI
* PLUS *number1 number2*
* POWER *number1 number2*
* QUOTIENT *number1 number2*
* RANDOM *number*
* REMAINDER *number1 number2*
* ROUND *number*
* SETRANDOM *number*
* SIN *number*
* SQRT *number*
* TIMES *number1 number2*

Defining and Editing Procedures

-> (Tab)
;
COPYDEF *word1 word2*
DEFINE *word list*
EDIT (ED) *word/list*
EDALL
EDNAME *word/list*
EDPROCS *WORD/LIST*
EDPROPS *word/list*
END
TEXT *word*
TO *word1 <word2 word3 … >*

Word, String, and Lists

= *object1 object2*
* BEFORE? *object1 object2*
* BUTFIRST (BF) *object*
* BUTLAST (BL) *object*
* COUNT *object*
* EMPTY? *object*
* EQ? *object1 object2*
* FIRST *object*
* FPUT *object list*
* GETPROP (GPROP) *word1 word2*
* ITEM *number object*
* LAST *object*
* LIST *object1 object2*
* LIST? *object*
* LOWERCASE *word/string*
* LPUT *object list*
* MEMBER? *object1 object2*
* NUMBER? *object*
* PARSE *string*
* PLIST *word*
* PUTPROP (PPROP) *word1 word2 object*
* REMPROP *word1 word2*
* SENTENCE (SE) *object1 object2*
* STRING *word/string1 word/string2*
* STRING? *word/string1 word/string2*
* UPPERCASE *word/string*
* WORD *word/string1 word/string2*
* WORD? *object*

Flow of Control/Logic

* AND *predicate1 predicate2*
  <predicate3 predicate4...>
CATCH *word list.to.run1*
  list.to.run2
GO *word*
GOODBYE
IF *predicate list.to.run*
  <list.to.run>
* NOT *predicate*
* OR *predicate*
OUTPUT (OP) *object*
REPEAT *number list.to.run*
RUN *list.to.run*
STOP
THROW *word*
TOPLEVEL
WAIT *number*

Input and Output

.MOUSEOFF
.MOUSEON
* ASCII *character*
* BUTTON?
* CHAR *number*
CLEARKEYS
CLEARTEXT
* DATE
FPRINT *object1 <object2 object3 …>*
FPRINT1 *object1 <object2 object3 …>*
* KEYSDOWN
* MOUSE
PRINT (PR) *object1 <object2 object3 …>*
PRINT1 *object1 <object2 object3 …>*
* RC?
* READCHARACTER (RC)
* READLIST RL
* READSTRING
* TICKCOUNT
* TIME

Debugging

CONTINUE (CO)
DEBUG
* ERROR
ERRORPAUSE
NODEBUG
NOERRORPAUSE
PAUSE
STEP *word/list*
* STEPPED
TRACE *word/list*
* TRACED
UNSTEP *word/list*
UNWATCH *word/list*
WATCH *word/list*
* WATCHED
* WHOCALLS *word*
* WHOUSES *word*

Filing and Managing Workspace

* CONTENTS
* FINDERFILES
GCOLL
* NODES
* BURIEDNAMES
* BURIEDPROCS
* BURIEDPROPS
BURYALL
BURYNAME *word/list*
BURYPROC *word/list*
BURYPROP *word/list*
EJECT *word/string*
ERALL
ERASE (ER) *word/list*
ERASEFILE *word/string*
ERNAME *word/list*
ERPROPS *word/list*
LOAD *word/string*
LOADPICT *word/string*
LOADTEXT *word/string*
* NAMES
* ONLINE
* PLNAMES
POALL
PONAME *word/list*
POPROPS *word/list*
POTS
* PREFIX
* PRIMITIVE? *word*
* PRIMITIVES
PRINTOUT (PO) *word/list*
* PROCEDURE? *word*
* PROCEDURES
RENAMEFILE *word/string1 word/string2*
SAVE *word/string*
SAVEPICT *word/string <word>*
SAVETEXT *word/string <word>*
SETPREFIX *word/string*
UNBURYALL
UNBURYNAME *word/list*
UNBURYPROC *word/list*
UNBURYPROP *word/list*
UNMOUNT *word/string*

Windows

* ACTIVEW
CLOSEWINDOW *word/string*
* INWINDOW [*number1 number2*]
MAKEWINDOW *word/string1*
word/string2
PRINTWINDOW *word/string*
SCREENTOW [*number1 number2*]
SCROLLRECT *number1 number2 number3*
number4
SETACTIVEW *word/string*
SETCURRENT *word/string*
SETWFONT *word/string*
[*word number*]
SETWPOS *word/string* [*word number*]
SETWSIZE *word/string*
[*word number*]
SETWWRAP *word/string predicate*
* WFONT *word/string*
* WINDOWS
* WINDOWTYPE *word/string*
* WPOS *word/string*
* WSIZE *word/string*
WTOSCREEN [*number1 number2*]
* WWRAP? *word/string* WTOSCREEN
[*number1 number2*]
* WWRAP? *word/string*

Stream I/O

SETPORT *port baudrate stopbits*
parity databits
* ALLOPEN
CLOSE *stream/list*
* EOF? *stream*
* FILELEN *stream*
* FPOS *stream*
OPENA *word/string*
OPENR *word/string*
OPENW *word/string <predicate>*
PRINT.TO.STREAM *stream object1*
<object2 object3 ...>
* READER
SETFPOS *stream number*
SETREAD *stream*
SETWRITE *stream*
* STREAM? *object*
* STREAM.INFO *stream*
* TERMINAL.INPUT
* TERMINAL.OUTPUT
* WRITER

Miscellaneous Commands and Special Words

ALL
DEPOSIT *number1 number2*
ERROR
EXAMINE *number*
FORMAT
AUTOLOAD
FALSE
LOGO.INIT
REDEF.PRIM?
TRUE
* VERSION

Index

" 83, 176
() 12, 83
.MOUSEOFF 174
.MOUSEON 174
< 105, 187
= 105, 187
> 105, 187
? 187
[] 12, 83
[SHAPE] 19
` 29

AND 187
Animation 97
Arithmetic
  with Logo 155
Arithmetic reporters 156
Arrow keys
  procedure to use 170
ASCII 170
ASK 19, 135
Assignment statements 175

BACK 6
Back quote 29, 83
BASIC 88
BEFORE? 190
BF 194
BK 6
BL 194
Blank lines 54
Body centered 181
Bottom up programming 49
Boxes
  as models for inputs 130
Brackets
  square 12
Bugs 47
BUTFIRST 57, 194
BUTLAST 194
BUTTON? 173

Cartesian reporters 185
Case statement
  creating 129
Categories of procedures 83
CG 5, 6
CHAR 55, 170
CLEAN 5
CLEARGRAPHICS 5, 6
CLEARTEXT 54
CLOSEWINDOW 69
CO 102
Color 18
Command 83
Conditions
  in IF 104
  testing 190
CONTINUE 102
Coordinates 181
COUNT 197
Counting
  using inputs 145
Cursor
  I-beam 4
  Logo Window 4

DATE 16, 173
DEBUG 102
Debug Window 102
Debugging 47
  by running each procedure 101
  using Pause and CONTINUE 102
  using PRINT 105
  using PRINT statements 102
  using TRACE 102
Defining shapes 21
Definite loops 103
Definition
  recursive 82
Delete key 6
Designing programs 49
DISTANCE 184
DISTANCEPOS 185
DOT 103
Dots 18, 125, 175
Drawing with the Turtle 4

EACH 133
ED 39
EDIT 39, 59
Editing
  immediate mode typing 16
Editor 39, 40
  long lines in 47
  tips for using 42
Editor Window 39
Embedded recursion 95, 151
  problems with 122
Empty word 194

EMPTY? 189
Enter 3
Equal sign 105
ER 60
ERALL 24
ERASE 37, 60
ERASEFILE 61
Erasing lines 7
Erasing procedures 37
ERNAME 119, 177
Errors 6

FALSE 187
FD 4
File Menu 7
Files
  erasing 61
  loading from procedures 119
  manipulating from procedures 119
  working with 61
Files and workspace
  relationship 62
Filling
  problems 27
FILLSH 26
FIRST 193
Flatten 202
Font
  changing 29
FORWARD 4
FPRINT 197
  and PRINT 197
FPUT 205
Functions
  in Logo 163

Global names 176
Grammar
  Logo 81
Graphics reporters
  in IF instructions 167
Graphing 182
Greater than 105
GREATER? 190

Harvey 84, 154, 193
HEADING 184
HIDETURTLE 9
HOME 18, 77
HT 10

IF 104
  problems with brackets 123
  structure 121
  structure of 105
  three input 121
  to stop loops 103
Immediate mode 39
Incrementing
  in graphics 150
  using inputs 145
Indefinite loops 93
Inputs
  procedure 125
  to pass values 128, 139
  using to increment 145
Interactive programming 115
Interactive reporters
  when to use 117
ITEM 198

Keyboard input
  keeping 139

LAST 194
LEFT 6
Less than 105
LESS? 190
Lines
  blank 54
  erasing 7
List 202
  flatten 202
  Logo 81
List processing 193
List-processing primitives
  distinctions 206
LIST? 188
LOAD 23
LOADPICT 6
LOCAL 178
Local names 176
Logo
  rules of 81
Logo list 81
Logo number 81
Logo punctuation 83
Logo string 81
Logo word 81
Long lines 26
  after IF 122
  in the editor 47

Looping....93
using procedure inputs....127
Loops
definite....103
indefinite....93
stopping....104
stopping with READCHARACTER....104
Lowercase
printing....53
LOWERCASE....57, 169
LPUT....205
LT....6

Main procedure....46
MAKE....175
when to use....175, 179
MAKE.WORD....32
MAKEWINDOW....69
Math
with Logo....155
Mathematical functions....158
MEMBER?....189
Menus
writing....129
Middle out programming....49
Mindstorms
Children, Computers, and Powerful Ideas....1
Mode
setting the pen....68
Modes
pen....63
Mouse....4, 173
Multiple turtles....133

NAMES....119, 176
explanation of scope....178
global....176
local....176
Naming
using OUTPUT....163
NOT....105, 187
Not equals....105
Number
Logo....81
NUMBER?....188
Numbers
reading....116

Objects
Logo....81
Operation....15
OR....187

Origin....181
OUTPUT....159
to create functions....163
to write tools....160

Papert....1
Parameter passing....142
Parentheses....12
Pascal....88
Passing parameters....142
Passing values....128
using inputs....139
Pausing....102
PD....6
PE....7
Pen
changing the pattern....14
changing the size....13
Pen mode
setting....68
Pen modes....63
chart of....68
PENDOWN....6
PENDOWN?....189
PENERASE....7, 63
PENMIX....66
PENMODE....67, 167
PENPAINT....7, 63
PENREVERSE....63
PENSIZE....167
PENUP....6
PICK....199
Pictures
loading from procedures....119
printing....16
saving....6
Pixel....17
PO....59
POALL....176
PONAME....177
POS....182
POTS....59
PPATTERN....167
PR....53
Predicate....106, 187
Primitive....2
PRINT....53
and FPRINT....197
parentheses with....56
used for debugging....101
used in debugging....105
PRINT1....55
Printing
case....53

Printing pictures 16
Printing procedures 40
Printing spaces 54
PRINTWINDOW 16
Procedure
  defined 40
  main 46
  sub 46
  super 46
  top level 46
Procedure inputs 125
Procedure names
  listing 59
Procedure tree 46
PROCEDURES 37, 59
  categories of 83
  describing 84
  formatting 121
  naming 47
  printing 40
  saving 40
  testing 47
  writing 39
Program
  stopping 94
Programming
  bottom up 49
  middle out 49
  polishing 50
  publishing 50
  top down 49
Programming style
  rules for 50
Programs
  designing 49
  interactive 87
Punctuation
  Logo 83
PX 63

Quotation mark 6, 176
Quoting
  lists 166
  special symbols 157

RANDOM 14
RC 87
READCHARACTER 87
  to stop a loop 104
READLIST 115
READWORD 116, 122
Recursion
  embedded 95, 151
Recursive call 94
Recursive definition 82
REPEAT 9
Reporter 15, 83
Reporters
  arithmetic 156
  Cartesian 185
  how to write 159
  predicates 187
RESULT 15
Resursion
  in looping 93
Return 3, 39
RIGHT 5
RL 115
RT 5

SAVE 40
SAVEPICT 6
SAVESHAPES 23
Saving
  a plan for 37
  establishing a plan 61
Saving pictures 6
Saving procedures 40
Scope
  dynamic 176
  of names 178
Semi colon 46
SENTENCE 56, 201
SETACTIVEW 70
SETH 183
SETHEADING 183
SETPENMODE 68
SETPENSIZE 13
SETPOS 182
SETPP 14
SETPPATTERN 14
SETPS 13
SETRANDOM 16
SETTFONT 30
SETTSTYLE 30
SETWFONT 72
SETWPOS 70
SETWSIZE
  and coordinates 181
SETXY 11, 181
Shape
  erasing 19
  filling with a pattern 25
Shape Edit 21
Shape Editor 21
  using 22
Shape Editor Window 21
SHAPE.EDIT 21

Shapes
defining 21
for animation 98
loading 23
modifying....98
saving 23
special pen position....35
turtle....17
using....17
SHOWTURTLE....10
SNAP....17
Snapped shapes
using....17
Spaces
printing....54
using....7
Square brackets....12
ST....3, 10
STAMP....17
STAMPARC....111
Stamping
arcs....111
areas....107
with borders....113
ovals....107
rectangles....109
Stamping the sturtle
steps to follow....18
STAMPOVAL....107
STAMPRECT....109
STAMPTEXT....11
STOP....130
in loops....195
Stopping a procedure....94
Stopping a program....94
Stopping loops....103
with READCHARACTER....104
String
Logo....81
Strings....29
with STAMPTEXT....56
Style
changing....30
Subprocedure....46
Superprocedure....46

Tab key....47
TELL....75, 133
problems with....78
TELL, EACH, ASK
differences....137
Text
from procedures....45
Text font
changing....30
normal....31, 32
Text fonts
problems with....31
Text in the Turtle Window....11
Text style
changing....30
Text styles
problems with....31
TFONT....165
that BUTTON?....189
TIME....173
Tools
using OUTPUT to write....160
Top down programming....49
Top level procedure....46
TOWARDS....185
TRACE....102
Tree
procedure....46
Triangle....4
TRUE....187
TSTYLE....165
Turtle
disappearing....19
Turtle shape
getting....19
Turtle steps....7
Turtle Window....4
sizing....69
Turtle, triangular....4
Turtles
more than one....75
multiple....75, 133
number of....76

UNTRACE....102
Upper case
printing....53
UPPERCASE....169
Using keyboard input....128
Using STOP
in looping....195

Variables
see inputs or names....176

WHO....78, 133

Window
Turtle
sizing....69
Windows
problems with....73
using....69
Word....201
empty....194
Logo....81
WORD?....188
Words
reading....116
Workspace....23, 40, 41
managing....59
Workspace and files
relationship....62
Wrapping....10
Writing procedures....39

XCOR....182

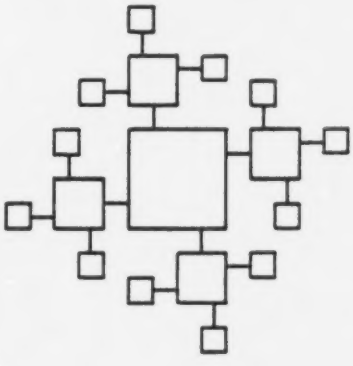
YCOR....182

Zero
and "oh"....4

Be Involved in ISTE's newest Special Interest Group, HyperSIG and receive

HyperNEXUS

Journal of Hypermedia and Multimedia Studies

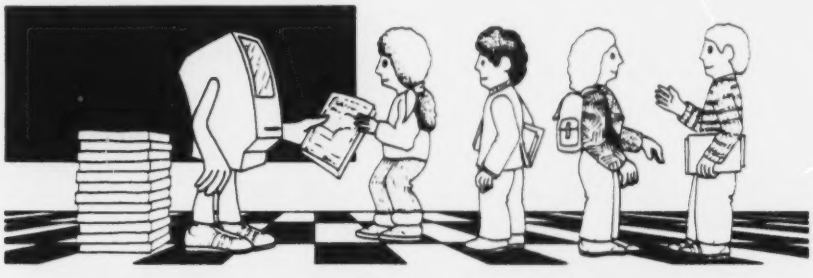


HyperSIG is ISTE's SIG for hypermedia and multimedia applications in education. HyperSIG collects and disseminates information on the use of hyper/multimedia in classrooms; sponsors conference presentations, activities, and meetings; and helps develop hyper/multimedia projects.

HyperNEXUS, published quarterly by HyperSIG, features articles on projects, lesson plans, and theoretical issues concerning the educational use of hypermedia and multimedia. Reviews of hardware, software and books help keep readers up to date in this rapidly changing field.

Contact ISTE for Membership Rates

ISTE,
1787 Agate St., Eugene, OR 97403-1923
503/346-4414



Works users—seek help.

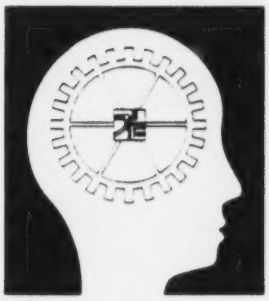
Practical and useful support is what you'll find in *Microsoft Works for the Macintosh: A Workbook for Educators* by Keith Wetzel. Step-by-step instructions thoroughly cover all the capabilities of MS *Works* including macros, spellchecking, and multiple column text and labels for Version 2.0. While you are learning, you'll create usable transparencies, letterhead, and other templates for use in your classroom or at home.

Microsoft Works for the Macintosh: A Workbook for Educators is available for both version 1.1 and 2.0.

Version 1.1 and 2.0 are available. Please specify which version you need when ordering.

ISTE
1787 Agate St., Eugene, OR 97403-1923
503/346-4414

Get help today. *Microsoft Works for the Macintosh: A Workbook for Educators* from ISTE.



Fantastic Journey Through Minds and Machines

by Michael Muir

Artificial Intelligence (AI) calls up visions of supercomputers and gigabytes of computer code. Michael Muir gives AI a human dimension and a practical application in the secondary classroom.

Even noncomputing readers will enjoy his entertaining chapter introductions on the important history and people of this technical and intellectual frontier. Muir explains the essential challenges of creating robots, games, natural language, machine intelligence, and expert systems.

For students and teachers of computer science, each chapter also includes a complete AI program in Logo (both *LogoWriter* and *Logo Plus* versions). The programs range in sophistication from a simple maze to an inference engine with natural language interface. The code is documented line by line, so that even novice programmers can follow the logic and development. Suggestions for program extensions offer expert-level challenges.

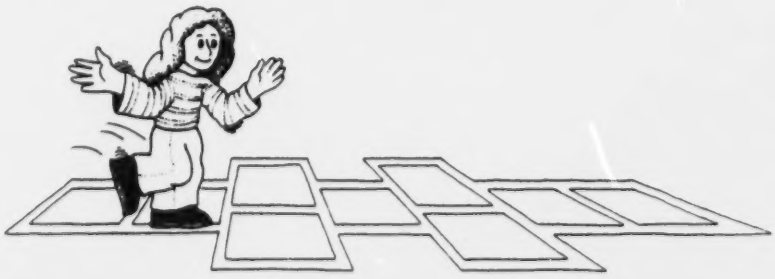
For pricing information or to order contact, ISTE, 1787 Agate St., Eugene, OR 97403-1923; ph. 503/346-4414.

We've taken the first step for you. Randy Boone pulls together the best research, position papers, product reviews and lesson plans for teaching writing.

Recent articles from *The Computing Teacher* and *The Writing Notebook* will take you yet another step closer to understanding the issues, problems, and solutions surrounding the writing process.

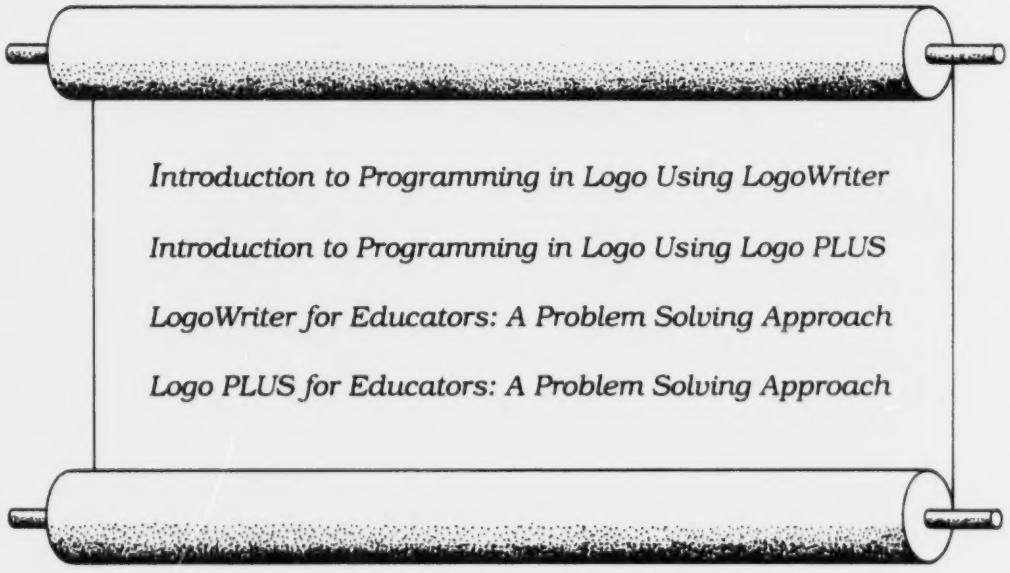
Use ***Teaching Process Writing with Computers*** in your university classes, inservice workshops, and K-12 classrooms. Your students will make great strides.

For pricing or to order, contact: ISTE, University of Oregon, 1787 Agate St., Eugene, OR 97403; ph. 503/346-4414.



Teach process writing—step by step.

Look at our Logo list!



Introduction to Programming in Logo Using LogoWriter

Introduction to Programming in Logo Using Logo PLUS

LogoWriter for Educators: A Problem Solving Approach

Logo PLUS for Educators: A Problem Solving Approach

Logo users at all levels benefit from these ISTE selections.

The *Introduction to Programming* books, written by Sharon Yoder, provide beginners with a Logo base to build on and experienced users with a reference to rely on. Both are excellent resources for teacher training or introductory computer science classes.

New from ISTE, *LogoWriter (Logo PLUS) for Educators: A Problem Solving Approach* takes Logo learning to new depths. The focus is entirely on learning and practicing general problem solving skills while using Logo. Great for beginning programming experience. Appendices include keystroke summaries, turtle shape pictures, and a quick reference card. Written by Dave Moursund and Sharon Yoder.

To order, contact:
ISTE, 1787 Agate St., Eugene, OR 97403-1923
503/346-4414

Add another brick to your Logo base.

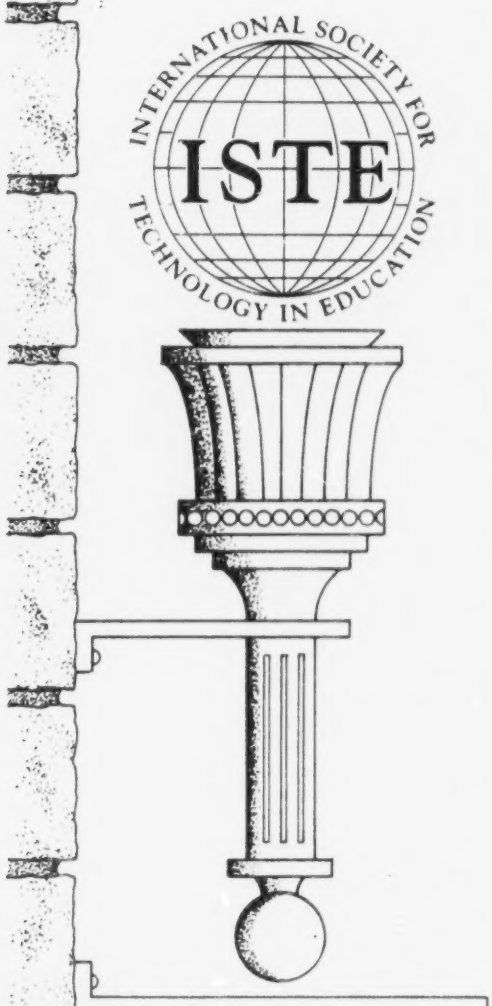
ISTE's Special Interest Group for Logo Educators (SIGLogo) provides you with a broad Logo foundation.

Build your knowledge with the latest information on Logo research, resources, and methods. Expand your Logo alternatives through the exchange of ideas, concepts, and techniques.

Both novice and experienced Logo users will find constructive uses in their SIGLogo membership.

Join SIGLogo now. Membership includes eight issues of *Logo Exchange*, the SIGLogo journal. Members are invited to participate in local, regional, and national meetings and to contribute to the flow of ideas through the *Logo Exchange*. SIGLogo membership rates are available by contacting ISTE.

ISTE, 1787 Agate St., Eugene, OR 97403-1923
503/346-4414



The ***International Society for Technology in Education*** touches all corners of the world. As the largest international non-profit professional organization serving computer using educators, we are dedicated to the improvement of education through the use and integration of technology.

Drawing from the resources of committed professionals worldwide, ISTE provides information that is always up-to-date, compelling, and relevant to your educational responsibilities.

Periodicals, books and courseware, *Special Interest Groups, Independent Study* courses, professional committees, and the Private Sector Council all strive to help enhance the quality of information you receive.

Rely on ISTE support:

- *The Computing Teacher* draws on active and creative K-12 educators to provide feature articles and carefully selected columns.
- The *Update* newsletter reaches members with information on the activities of ISTE and its affiliates.
- The *Journal of Research on Computing in Education* comes out with articles on original research project descriptions and evaluations, the state of the art, and theoretical essays that define and extend the field of educational computing.
- Books and courseware enhance teaching materials for K-12 and higher education.
- Professional Committees develop and monitor policy statements on software use, ethics, preview centers, and legislative action.
- The Private Sector Council promotes cooperation between educational technology professionals, manufacturers, publishers, and other private sector organizations.

It's a big world, but with the joint efforts of educators like yourself, ISTE brings it closer. Be a part of the international sharing of educational ideas and technology. Join ISTE.

Join today, and discover how ISTE puts you in touch with the world.

Basic one year membership includes eight issues each of the *Update* newsletter and *The Computing Teacher*, full voting privileges, and a 10% discount off ISTE books and courseware.

Professional one year membership includes eight issues each of the*Update* newsletter and *The Computing Teacher*, four issues of the *Journal of Research on Computing in Education*, full voting privileges, and a 10% discount off ISTE books and courseware.

ISTE, University of Oregon,
1787 Agate St., Eugene, OR 97403-9905
ph. 503/346-4414